Implementation Effort and Parallelism - Metrics for Guiding Hardware/Software Partitioning in Embedded System Design

Abildgren, Rasmus

Publication date:
2010

Document Version
Accepted author manuscript, peer reviewed version

Implementation Effort and Parallelism – Metrics for Guiding Hardware/Software Partitioning in Embedded System Design

Ph.D. Thesis

Rasmus Abildgren

Department of Electronic Systems
Technology Platform Section
Aalborg University
Fredrik Bajers Vej 7
9220 Aalborg Ø, Denmark

Implementation Effort and Parallelism – Metrics for Guiding Hardware/Software Partitioning in Embedded System Design
Ph.D. thesis

ISBN: 978-87-92328-42-7
December 2010

Department of Electronic Systems
Aalborg University
Fredrik Bajers Vej 7
DK-9220 Aalborg Ø
Denmark

This thesis was typeset using LaTeX 2_ε.

Abstract

Implementation Effort and Parallelism – Metrics for Guiding Hardware/Software Partitioning in Embedded System Design

Hardware/Software partitioning is the task of deciding onto which architectures the algorithms of an embedded system should be implemented. Many criteria influence this decision. This thesis focuses on two important parameters targeting the implementation effort and the algorithm architecture affinity toward FPGA implementation.

In the introductory part of this thesis, we give an overview of the challenges faced by the designers in the industry, and identify hardware/software partitioning as one of the most critical phases in the design trajectory. We discuss typical parameters, the accuracy of different estimation techniques, and the many influencing factors when devising metrics for guiding the partitioning phase. In the last part of the introduction we give an overview of the research efforts related to implementation effort estimation, execution time estimation, and the framework and methodologies which use these techniques.

In the main part of the thesis, which is composed of four papers, we present our contributions to the topic of HW/SW Partitioning guidance. In Paper A and Tech Report B we deal with the estimation of the implementation effort for implementing algorithms onto FPGAs. In Paper A we present a metric-based approach for estimating the hardware implementation effort for an application in relation to the number of linear-independent paths of its algorithms. We show that a relation between the paths and the needed effort exists. In Tech Report B we further investigate the issue of accurate implementation effort estimation with respect to real-time constrained systems. Here we measure the implementation hardness of algorithms with respect to the given time constraint.

In paper C, we extend a promising static partitioning approach with a parallelism measure to better handle the estimated performance of FPGAs. More specifically we improve the affinity metric with parallelism detection, which is essential when characterising an algorithms potential for FPGA implementation, and this completely changes the FPGA affinity score for some algorithms. And Finally in paper D, we include a study which clearly shows the differences between static estimation techniques and the actual performance of a system generated through system level tools.

Synopsis

Implementeringstid og parallellitet - Metrikker til guidet opdeling af hardware/software i indlejrede systemer

Opgaven med at bestemme, på hvilken arkitektur en given algoritme skal implementeres, kaldes hardware/software partitionering. Mange kriterier skal tages i betragtning ved et sådant valg. Denne afhandling fokuserer på to indflydelsesrige parametre: implementeringstid og algoritme-arkitektursamhørighed set i forhold til FPGA implementering.

I introduktionen til denne afhandling gives et overblik over de udfordringer, som designere i industrien møder, og hardware/software partitionering identificeres som en af de mest kritiske faser i et moderne designforløb. Vi diskuterer typiske parametre, forskellige estimeringsteknikkers nøjagtighed samt de mange andre faktorer, som påvirker de metrikker (måleenheder), der kan guide partitioneringsfasen. I den sidste del af introduktionen gives et overblik over forskningens nuværende stadie inden for implementeringstid, eksekveringstidsestimering samt en kort præsentation af frameworks og metoder, der benytter disse teknikker.

Hoveddelen af denne afhandling består af 4 videnskabelige artikler. I disse præsenterer vi vores bidrag inden for guidning af HW/SW partitionering. I artikel A og teknisk raport B behandles estimeringen af implementeringstid i forbindelse med implementering af en algoritme på en FPGA arkitektur. I artikel A præsenteres en metrikbaseret fremgangsmåde til at estimere hardware- implementeringstiden for en applikation i relation til antallet af lineært uafhængige stier i dens algoritme. Vi viser desuden, at relationen mellem uafhængige stier og den nødvendige implementeringstid eksisterer. I teknisk rapport B behandles nøjagtigheden af implementeringstidsestimater i henhold til realtidssystemer og de tilhørende problemstillinger. Her måles sværhedsgraden for en imple-

mentering af en algoritme med hensyn til et givet eksekveringstidskrav.

I artikel C udvider vi en lovende statisk partitioneringsfremgangsmåde med et mål for den iboende parallelitet i en algoritme. Dette for bedre at kunne estimere dens mulige ydelse på en FPGA. Mere specifikt forbedrer vi samhørighedsmetrikken med detektion af parallelitet, hvilket er essentielt for en algoritmes potentiale på en FPGA. Dette forandrer fuldstændigt algoritme-FPGA-samhørighedsværdien for nogle algoritmer. Til slut i artikel D inkluderer vi et studie, som klart viser forskellen mellem statiske estimeringsmetoder og den reelle ydelse af systemer, genereret via system level tools.

List of Papers

The main body of this thesis consists of the following papers:

[A] R. Abildgren, J-Ph. Diguet, P. Bomel, G. Gogniat, P. Koch, and Y. Le Moullec, "A Priori Implementation Effort Estimation for HW Design Based on Independent-Path Analysis", in *EURASIP Journal on Embedded Systems*, Vol. 2008, pp.1–12, September 2008.

[B] R. Abildgren, J-Ph. Diguet, G. Gogniat, P. Koch, and Y. Le Moullec, "Real-Time Aware Hardware Implementation Effort Estimation", Technical Report, *Aalborg University*, 2010.

[C] R. Abildgren, A. Šaramentovas, P. Ruzgys, P. Koch, and Y. Le Moullec, "Algorithm-Architecture Affinity – Parallelism Changes the Picture", in *Proc. of the Design and Architectures for Signal and Image Processing Workshop*, 2007, pp. 1–4.

[D] A. Šaramentovas, P. Ruzgys, R. Abildgren, and Y. Le Moullec, "HS-DPA Design Space Exploration and Implementation Guidance with Design-Trotter", in *Proc. of the 6th IEEE International Conference on Information, Communications and Signal Processing*, 2007, pp. 1–5.

The following additional paper has been published by the author during the Ph.D. studies, and was selected as one of best 15 papers at the conference. However, the paper has a significant overlap with paper A, and is therefore not included in this collection.

[1] R. Abildgren, J-Ph. Diguet, P. Bomel, G. Gogniat, P. Koch, and Y. Le Moullec, "A Method for A Priori Implementation Effort Estimation for Hardware Design", in *Proc. of IEEE International Conference on Electronic Design*, pp.1–6, December 2008.

Preface

This thesis is written as a partial fulfilment of the requirements for the Ph.D. degree from Aalborg University. The work was carried out during the period October 2005 to April 2010 and was supported by ETI A/S[1].

I would like to thank my two supervisors, Yannick Le Moullec and Peter Koch, for their advice, encouragement and patience, and for giving me the opportunity to pursue a Ph.D. I also thank all my colleagues at Aalborg University for making it a greater experience to be a Ph.D. student. I am also very grateful to Jean-Philippe Diguet and Guy Gogniat at LabSTICC at Université de Bretagne-Sud, Lorient, France who welcomed me in the lab in the autumn 2007, spring 2008 and summer and autumn 2009. I feel that my stays in Lorient made a big difference in forming this Ph.D.

A special thanks also goes to Kods Trabelsi and Marwa Kanso for extended assistance with coping with the French bureaucracy, great hospitality, and for making many weekends in the lab fun and enjoyable. I would also thank Maleeha Kiran for proofreading some texts in the very last minute.

From ETI A/S I would like to thank the former Accelerated Processing group who provided me with a lot of help and assistance with achieving data for my experiments, especially René Bastrup Knudsen and Karl-Ejner Christensen who had the job to read all my raw data and drafts for company approval.

Last but not least, I thank my family and friends for encouragement and support, especially Hans Laurberg, Jesper Højvang Jensen and Jesper Abildgaard Larsen for lot of encouragement in the difficult times.

[1]ETI A/S, Nørresundby, Denmark, is a company which provides diagnostic and data analysis products to law enforcement agencies and telecommunication providers.

There is a glossary section in the end of the introduction. This replace many definitions in the introduction, since I usually find it very easy to deduce from the context what the different terms mean, and having many definitions in the introduction disturbs the reading flow.

Contents

Introduction

In this chapter we introduce the reader to the topics addressed in this thesis. We begin by introducing the challenges faced when designing modern embedded systems. In Section 2 we present the typical design steps needed when designing heterogeneous multiprocessor systems. One of the design steps is refered to as hardware/software partitioning, this is further addressed in Section 3. This step is the domain in which the work of this thesis is carried out. This is followed by Section 4 that summarises the state-of-the-art related to methods for hardware/software partitioning and the criteria for design decision, i.e. design space exploration and the underlying metrics. Subsequently, Section 5 presents our contributions to the field. Lastly, we have included a taxonomy (Section 6) that defines the main terms of the field and the way they have been used in this thesis.

1 Background

When developing embedded system(s) products, e.g. a mobile phone, there are specific demands and needs from the market, i.e. end users, sales people, operators, etc., and pressure from competitors which encourage companies to produce innovative and/or increasingly advanced products. New products are therefore often more complex in their design than the previous ones but are yet released with an ever increasing frequency. As a result of the constant progress of very-large-scale integration (VLSI) technology (e.g. Moore's Law[2] [1]), it is possible, to some extent, to cope with these needs.

However, technological progress does not come without related prob-

[2]Well knowing that it is an observation and not a law

lems. When systems become larger and more complex, their design becomes more complicated and challenging. This results in more time consuming development phases. As frequency of release of new products increases, a conflicting situation arises where it is not possible to get a product which needs a longer development time to the market in less time, unless something in the design strategy is changed. Such a change could either involve allocating more human resources for the development phase; or regarding the design process, reusing parts of existing designs, raising the level of abstraction at which the design is conducted, or a combination of any of the above. Each of the above changes to the design strategy will come with their own set of issues and problems which will have to be addressed.

Allocating more human resources in order to reduce the development time is a very expensive solution. Moreover, there is a certain limit [2] to how efficient this can be as, it is not always possible to break down the problem into separate development tasks and increasing human resources require a lot of communication and management, an overhead which decreases the efficiency of the development team.

Reuse parts of existing designs, whether they are in-house IPs or from an external vendor, could reduce the development time drastically but designing a unique and innovative product makes it difficult to find suitable existing building blocks. Furthermore, the integration cost and the quality of the IPs, especially those from external vendors, are non-negligible factors which have to be considered.

Thus, in many cases the most promising strategy is to raise the level of abstraction at which design is conducted [3], i.e. making design decisions on a higher level without having to worry about implementation details. This could be done by using a higher level programming language such as C instead of assembly, or Behavioural VHDL instead of RTL (Structural VHDL). In addition e.g. design decisions about the hardware platform could be done earlier in the design process.

To take full advantage of higher levels of abstraction, they should be used for both the methodology and supporting tools [4]. In this way the tools will take care of the lowest levels of abstraction (e.g. assembly or RTL). This enables the design team to focus on handling increasing complexity of design which involves:

- Removing unnecessary design details;
- Enabling earlier design decisions;
- Enabling easier identification of reusable parts.

Although many existing tools are able to convert a higher level description into e.g. RTL, they have not yet been widely adopted for a number of reasons [5], including:

- Licensing costs are high, especially for tools with proven functionality.
- Introducing these tools implies significant changes to the overall design methodology, including the way the development tools are used. Therefore, extra investments are required for educating the designers.
- Although tools have matured over the last couple of years, there is still a lack of supporting libraries and IPs for many application domains. Moreover, many of the tools do not cover the entire design trajectory, which requires buying several tools and interfacing them.
- For many practical applications, real-time execution is required. Even though these tools include high level synthesis features, their ability to transform algorithms to exploit the architectural features is still limited. Therefore, designers still have to carry out low level implementation for the critical parts of the application, in order to meet real-time constraints.

Many companies do not use higher level synthesis tools due to the reasons stated above, but they can still raise the level of abstraction in the methodology and thereby allow the design teams to take design decisions at higher levels (i.e., earlier in the design trajectory). An example of this is system partitioning.

Considering today's complex embedded systems, one of the major challenges for designers is to have a complete overview of the large systems they have to develop. These systems often have stringent requirements for their platforms, which typically are of the heterogeneous multiprocessor system type. Working with multiple processors and a heterogeneous

architecture makes it necessary to decide where the different parts of the system should be implemented. This task is called system partitioning and has a significant influence on the rest of the design as well as the design time. The choice is often between implementing a particular functionality on either a general purpose CPU, a digital signal processor (DSP), or on a field-programmable gate array (FPGA). Traditionally, a general purpose CPU is not optimized for any particular application, while both the architecture and instruction set of a DSP is optimized for data processing. An FPGA can take advantage of the inherent parallelism of algorithms, but is slow for sequential processing. Changing the partitioning decisions later in the design trajectory will be very time consuming as a significant part of the design trajectory will have to be repeated. Therefore it is of great importance to analyse the different parts of the system functionalities with the purpose of determining their possible performance and cost on the different processing elements which are present in the system. This task is called Design Space Exploration, or DSE, and we will return to this later in subsection 2.2. The choice of partitioning is especially critical for hard real-time constrained systems where the high level synthesis tools are typically not mature enough to produce efficient code.

2 General Design Trajectory

Designing embedded heterogeneous multiprocessor systems requires an extended analysis compared to "simpler" systems. Many researchers e.g. [4, 6–10], as well as tool vendors, e.g. [11], have proposed different design trajectories for addressing this problem. Even companies usually modify or create their own trajectories to fit their needs, see e.g. [12].[3] Typically these trajectories target different types of applications, which involves both benefits and drawbacks. They are composed of many individual steps which correspond to specific design activities and are summarized in the general design trajectory described below as well as illustrated in Fig. 1.

Analysing from a general point of view, the design trajectory consists

[3]It is our experience that companies' design trajectories are not well founded in theory but more based on experience and intuition.

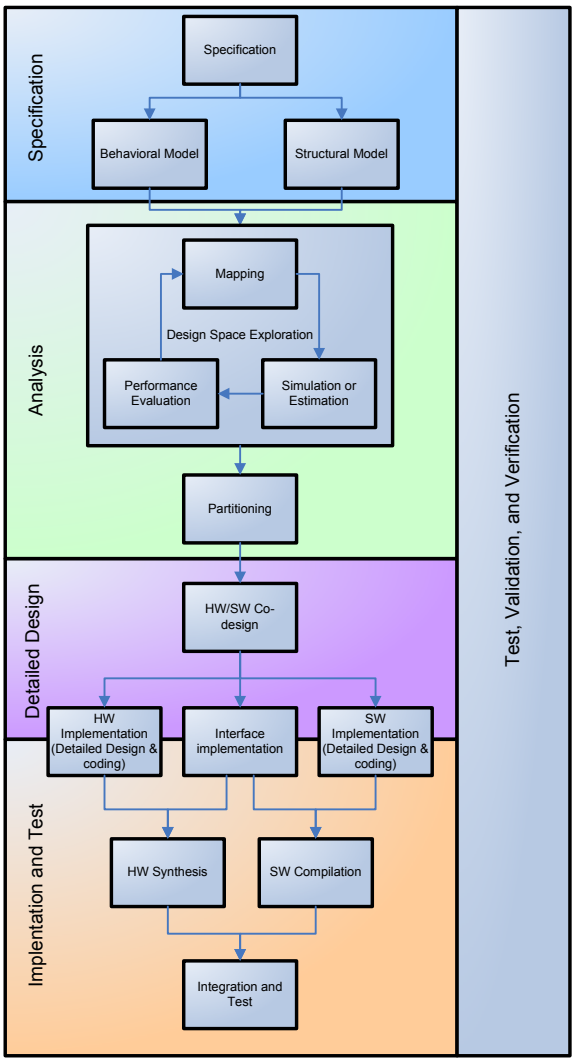


Fig. 1: The general design trajectory for embedded multiprocessor systems. The illustrated design trajectory summarises the main contents of a design trajectory involving a design space exploration in the analysis.

of 5 major phases: Specification, Analysis, Detailed Design, Implementation, and Test, Verification and Validation. The four first ones form a flow from specification to the final product, whereas Test, Verification and Validation needs to be conducted concurrently with the flow.

2.1 Specification: Capturing Functionality, Constraints, and Other Requirements

Every entrance to the design trajectory begins with the specification of the product. Such a specification can be conducted in many ways and at many levels of details. The specification typically includes a description of the behaviour of the product as well as a description of its structure (architecture). The behavioural description describes in details what the system should do at any given time. To support this description, many different behavioural models exists, e.g. FSM, Petri Nets, SDF. Furthermore, there exists many different behavioural specification languages which support one or more of these models. The usual specification languages are UML diagrams, the SpecC , SystemC, C++ or C languages, as well as other languages created specifically for this purpose also exist. Although C is not designed for describing the behaviour of hardware systems in detail, it is still the most dominating language for executable specifications.

Requirements for the product can be separated into several domains. Requirements and constraints can be in the domains of I) application requirements, II) methodology requirements, e.g. specific methods or approaches for designing the products, and III) platform requirements, e.g. a need for using a specific processor or FPGA. Most works clearly focus on the application requirements (which we in the remainder of this thesis will call the functionality or the behaviour), and to some extend on the platform requirements. Many do not consider methodology requirements although this is crucial for enabling companies to address the time-to-market factor.

2.2 Analysis

The specification phase is followed by the analysis phase. The purpose of the analysis phase is to gain knowledge which will enable the designer to select a promising design solution from many possible solutions, before

actually designing it in detail. This solution for the entire system should be chosen so that it has a high likelihood of fulfilling the requirements. This is particularly true for the HW/SW partitioning decision which have a large influence on the final system's performances. Many different solutions which fulfill the requirements exist. One can visualise this analysis as the task of finding the path which leads from the specification to a solution that fulfills the requirements. The challenge is not only to identify the path which will lead to an acceptable solution, but also to identify those which would lead to inefficient solutions, so that the design space can be pruned.

In order to identify those paths, it is necessary to evaluate the properties of the different solutions and to navigate between those. As mentioned earlier, this task is called design space exploration (DSE), and consists of two orthogonal activities [13]: I) generate, according to the desired exhaustiveness level, different solutions on the basis of the behavioural specification and II) evaluate the performance (according to a cost function) of the individual solutions.

Populating the Design Space

The problems related to the generation of the solutions for a heterogeneous multiprocessor architecture raises several issues. First of all, the architecture is not always "given" and needs to be defined (i.e. selecting the type and number of processing elements, also called allocation, as well as the communication topology). Second, the algorithm(s) must be mapped (which includes, among others, binding and scheduling) onto the architecture. These processes are iterative since they influence each other. Each of these design activities can be influenced by a large number of parameters, which combined together create a plethora of possible solutions.

The generation of different solutions in such a solution space is a multidisciplinary challenge. In the field of design space exploration different techniques are used. The popular ones are:

Genetic Algorithm: Using the Genetic Algorithm approach to the design space is typically populated by using random mutation, and/or crossover and evaluated according to a fitness function which guides

the generation of the next generation of solutions. Genetic algorithms have gradually emerged from the domain of evolutionary computing of biological systems and was first popularised by Holland [14] who among others formalised a framework for prediction of the quality of the next generation. For design space exploration, genetic algorithms are used among others by [15–18].

Simulated annealing: In simulated annealing, solutions which have been evaluated as fit are replaced by nearby solutions with the expectation to identify a local optimum. In order to find the global optimum, heuristics are used to perturb the selection process. Simulated annealing was independently developed by Kirkpatrick et al. [19] in 1983 and Cerny [20] in 1985. For design space exploration, simulated annealing is used among others by [21].

and more exotic methods including some of the following:

Random generation: Random generation is typically used either to generate the initial population (e.g. in genetic algorithms) or to introduce diversity when renewing the population. It can also be used as an independent method for the generation of solutions.

Bayesian Belief Networks: Bayesian Belief Networks (BBN) for populating the design space was first proposed in [22]. Using the BBN, they are able to model the development in the partitioning, when deciding to bind certain parts of the algorithm to an architecture. The performance information for every evaluation of the partitioning needs to be fed back into the network, and thereby develop the knowledge of the different partitionings, to find the one best suited. [23] has also done some experiements with this method.

Particle Swarm Optimization: In Particle Swarm Optimization (PSO), optimisation is done by having a population of candidate solutions called particles which are moved around in the design space according to a formula. The direction of the particles are guided by the best found positions in the design space. For every iteration the best found positions are updated as better positions are found. The method where original developed by Kennedy and Eberhart [24]. This method is used for DSE in [25].

Evaluation and Comparison of Solutions

For each and every generated solution, an evaluation has to be performed. With a very large design space this is not a negligible task. It has been shown [26] that exploring the corresponding unconstrained design space is $\mathcal{NP}$complete. Recent theoretical works on this topic include [27]. When the design is constrained by e.g. real-time requirements, the size of the design space is still so large that the search complexity is near $\mathcal{NP}$complete. Therefore, in order to keep the search time realistic, heuristic methods are used in practice.

The heuristic approaches simplifies the problem by introducing models which are often experience-based. Heuristics enable faster design space exploration at the expence of accuracy (the optimal solution might not be found). The heuristic approaches for design space exploration generally falls into two categories: simulation based and static analysis based. The former relies on the execution of one or several models to predict the behaviour and performance of the system (this can be carried out on all levels of abstraction). The latter consists in extracting relevant properties of the system in order to predict its performance. This is further discussed in Section 3.

The criteria for selecting the right solution(s) are set by a cost function which should be minimised. The cost function is formed by a set of relevant cost-parameters, on the basis of the project requirements and constraints. Typically, cost-parameters in the cost function are Execution Time (T), Area (A) and Power/Energy (P/E), but other parameters can be considered. All generated solutions are evaluated and their cost are computed. The solution(s) which minimise the cost function are the one which are expected to have the best fit with the requirements and can then be selected for further development. When selecting an evaluated solution, a certain partitioning of the system is chosen. This represents one of the major design decisions in the design trajectory. This topic is where we have carried out research and we discuss it in further details later in this thesis.

2.3 Detailed Design, Implementation, and Test

Fig. 1 shows a complete design trajectory. However, since the focus of this thesis is mainly on the specification and analysis steps, the last three steps

(Detailed design, Implementation, and Test) are only briefly discussed in this section.

Once the the partitioning of the system has been determined, it is now possible to proceed with the detailed design of the subsystems. Working with a heterogeneous multiprocessor system implies that many tasks partitioned on different processing elements need to interact with each other. It is therefore important to take care of the interaction between the hardware and software parts and how they should be interfaced. This can be done using a co-design approach in which the communication interfaces are specified in an architecturally transparent way.

After this, it is possible to proceed with the HW and SW implementations (including architecture specific interface) which we, in this thesis, consider as the tasks of designing the functionality in detail as well as carrying out the actual coding. Depending on the behavioural model of the specification, the process of carrying out the implementation can be either fully automated, semi automated/manual, or manual. In the case of semi automated or manual implementation, there is typically a great deal of interaction between the detailed design stage and the coding.

Once the codes are synthesised (HW) and compiled (SW) it is possible to proceed with the integration of the entire system and to perform the end test. In the ideal case, the criteria for passing the end test should be easily met since the system has been tested, verified and validated concurrently with the design flow, and the partitioning of the system has been done based on performance estimates obtained using appropriate models.

3 Guiding the HW/SW Partitioning Phase

3.1 Background

Being able to decide onto which processing elements an algorithm or sub-part of an algorithm should be implemented is typically based on some criteria which are specified in the cost function, with the goal of minimising it.

Let's consider a small example. An algorithm needs to be implemented on a heterogeneous multiprocessor platform consisting of two processing elements. The designer has to decide onto which processing

element the algorithm should be implemented. This depends on many different criteria. If execution time constraints exist, the execution time of the algorithm on the processing element is an obvious criteria. The affinity between the algorithm and the architecture of the processing element (e.g. special purpose instruction which can speed up the execution) should be decisive. However, if the algorithm requires a lot of communication with other algorithms in the system, the interprocess communication (IPC) will have a large impact on the execution time as well. Although one processing element is very well suited for executing the particular algorithm, it might be more efficient to map it onto the other processing element if the other algorithms are mapped there. In that way the cost of IPC can be minimised. Similarly, it might be more efficient to map the algorithm onto a processing element of which the implementation trajectory is faster and easier, if the development time is a critical issue.

As illustrated in this example, HW/SW partitioning is a non-trivial task since the individual criteria may influence each other and local decisions might lead to non-optimal final solutions.

3.2 Performance and Process Cost Functions

The importance of the cost-parameters varies from project to project and company to company. For example, in high volume projects, the financial cost (i.e. price) of the end product will have an influence while the Non Recurring Engineering (NRE) cost is less critical as it will be spread across many instances of the same product. Another example could be a system which deals with hard real-time constraints, for which the execution time is the most crucial parameter to optimise. Weighting the parameters which are in the cost function is based on their importance for each project.

As earlier mentioned, typical cost-parameters are execution time, area in terms of memory footprint and/or use of gates, and the power or energy consumption of the algorithm once mapped onto the architecture. All these reflects properties of the algorithm/implementation which can be used when estimating the output properties of the implementation, which can be related to the application requirements from the specification.

However, considering these technical properties might not be sufficient to evaluate the solution towards the entire set of requirements (Applica-

tion, Methodology and Platform as stated in subsection 2.1). For example, the non recurring engineering cost plays an important role for low volume products, which typically is the case for Small and Medium sized Enterprises (SMEs). Based on our observations as well as others [28], the main criteria for selecting and/or constructing a suitable architecture for many of these types of projects is related to the number of development hours that need to be allocated to the project.

In small as well as larger companies, the availability of the developers' resources can also be a criteria in deciding who is going to carry out the implementation. It can therefore be of vital importance to know the amount of time it will take to implement an algorithm on different architectures, together with knowledge of their performance, to determining the appropriate architecture for an algorithm.

We can therefore classify the output parameters which are used to partition the system into two different classes:

- Performance oriented parameters are parameters which reflect the performance properties of the final implementation such as execution time, area, power;
- Process oriented parameters are parameters which focus on the methodological aspects of the implementation such as reuse of IPs and implementation effort.

3.3 Analysis Classes and their Accuracy

In order to extract the parameters for the cost function from the algorithms, an analysis of the algorithm needs to be performed. The input to the analysis is the behavioural specification of the algorithm. The analysis can be performed at different levels of abstraction depending on the desired accuracy of the output.

In general we distinguish between two different analysis types: dynamic analysis, where data or a representative set of data are used for performing the analysis, and static analysis where only the algorithm itself is analysed. Dynamic analysis includes [13] Cycle-accurate simulation, trace-based simulation, instruction set simulation, and system-level simulation.

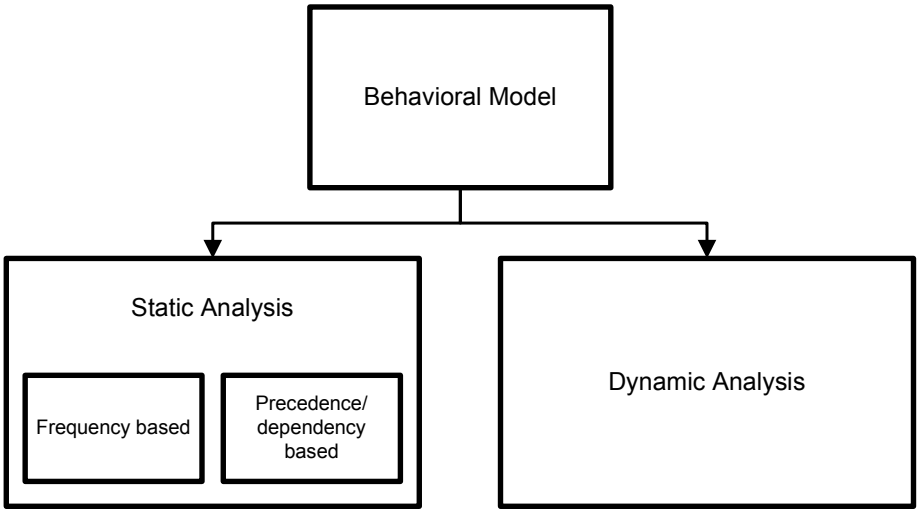


Fig. 2: This figure shows the relation between the different types of analysis (static as well as dynamic) and the different types of output parameters. The interesting part of this figure is that when it come to parameters estimating parts of the process, no method (to the best of our knowledge) so far includes dynamic information.

In static analysis, different categories of approaches also exist. In this thesis we devide static analysis into two sub-categories: analysis where the precedences between the operations are taken into account, and analysis where only the frequencies of occurrence of the different elements are counted. The latter is typically a textual analysis of an algorithm where for example the number of operators is counted, whereas in the case of the former, an analysis of the algorthim is conducted by measurement done using graphs.

Trade-off Between Dynamic and Static Analysis

Choosing the right estimation technique (dynamic or static) can be of vital importance for the partitioning phase in relation to the criteria. The difference between dynamic and static analysis and the arguments for choosing one or the other is therefore worth discussing. The static analysis is considered to be more light-weight and fast, but also less accurate compared to the dynamic analysis. In current literature that we have found, it is therefore surprising not to see more discussions of this topic.

In what follows we therefore elaborate on this difference.

Obtaining the performance estimates requires an analysis of how the behavioural specification can be executed on the architecture. To do so, a model of the architecture and an execution model are applied to the specification. The accuracy of the estimates depends on the accuracy of these models. E.g. a cycle accurate architectural model is typically more complex and will take longer time to evaluate than a transaction level model of the same architecture. On the other hand, the estimates of the former will be more accurate than those of the latter. The models range from physical level (e.g. transistors modelled with differential equations) to system level models consisting of estimated properties of the underlying architecture. Similarly, the execution models range from electric signals to simply counting the number of operations. Thus, there is no doubt that the complexity of the analysis increases when the desired accuracy and precision increase.

Information about the actual trade-off between accuracy and the complexity of the different estimation techniques is rarely found in literature. An attempt to compare the different classes is done in the survey by Gries [13]. From the survey we can deduce that the typical estimation error for static methods starts from 9%. For the dynamic cases the minimum error ranges from 0% to 9% for cycle-accurate simulation and for system-level simulation, respectively. In between these ranges are located the trace-based analysis (error starting at 1.5%) and instuction set simulations (error starting at 4%) error estimations. The accuracy of the estimates depends on the level of detail used in the underlying model, which is reflected in the computational complexity of the estimation process. Although the general trend is that the complexity of the analysis increases with the accuracy, some of the accurate methods can deliver their estimates at the same cost as less accurate methods. In general the trend is that improving the accuracy by 3 percentage points incurs a complexity increase of one order of magnitude.

Choosing an analysis method for a particular cost-parameter is not only dependent on the accuracy of the methods but also on the nature of the underlying factors such as their sensitivity to variations in input data. If a factor does not change with the input data, a static analysis should be able to deliver the same accuracy as a dynamic one. For example considering estimation of the size of a project, every corner of the project

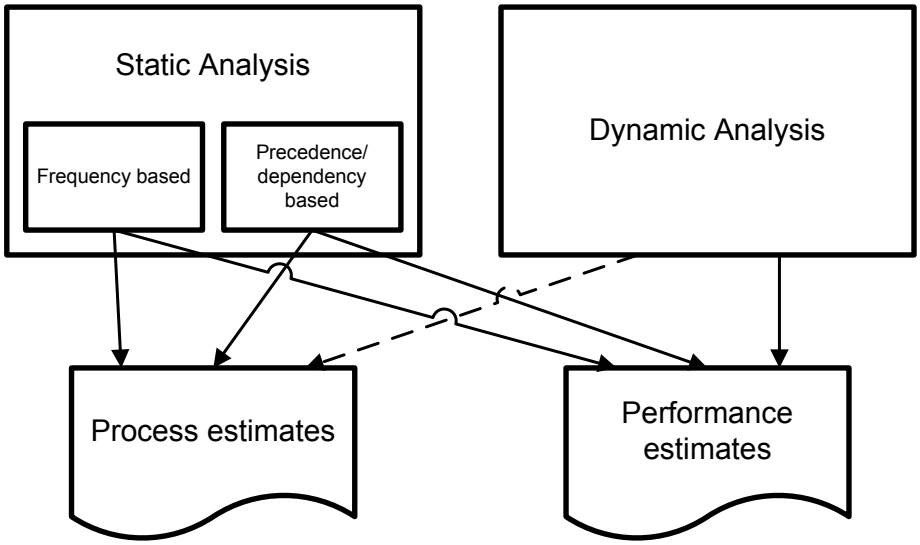


Fig. 3: The relation between the analysis classes and the metrics proposed in this thesis.

usually[4] needs to be implemented and therefore the size will not change with the input data.

Considering the performance oriented cost-parameters we see that both static and dynamic methods are used. This is because the accuracy as well as the computational complexity plays a role in the evaluation process. For the process oriented parameters, we have on the other hand not seen any methods which are based on a dynamic approach. One possible reason is that the process oriented parameters are less data sensitive than the performance oriented ones. The connection between the different classes of analysis and the contributions of this thesis are illustrated in figure 3.

3.4 Implementation Effort

The goal of estimating the implementation effort is to get an estimate of the amount of time involved in doing the implementation.

In order to estimate the implementation time, it is necessary to have an overview of all the factors and conditions that influence the imple-

[4]This assumes that the behavioural model does not include any unnecessary paths

mentation effort. We will refer to these factors as influencing parameters in this thesis. In this section we will shortly illustrate the variety of influencing parameters that needs to be taken in to account.

There are several different ways to categorize the influencing parameter, here we use the following three groupings:

- Human;
- Algorithm;
- Architecture.

Going in more detail about these categories we present which elements they involve.

The human category deals with all the personal factors (such as developer skills, team interaction) that influence the implementation process. An non-exhaustive list of the factors belonging to "human" category is shown below:

Designer multitasking overhead: The number of concurrent project a developer is working on influences his or her focus. Shifting between many task during the course of the the day may lead to longer implementation time. [29]

Domain experience: The designer's level of knowledge and experience from working with the problem domain may assist him or her in the design, and thereby shorten the implementation time.

Tools experience: If the designer is unfamiliar with the tools used in the project, his or her ability to exploit the capacity of the tools will be limited and the implementation time will be prolonged.

Skills: The designers level of skill needed for solving the problem also plays a role. E.g. if he/she needs to spend time in familiarising himself/herself with the maths of the algorithm.

Social atmosphere: The quality of the social interactions between the team members will impact their ability to cooperate.

Cooperation: The degree of cooperation between the designers within the project team is an important factor. When working as a team, activities need to be more coordinated, which will require extra time [29].

The second category of influencing parameter, the algorithm itself, deals with the specifications of the algorithm (such as the size and the number of constraints) that will influence the implementation time, no matter when or where the algorithm is implemented. These influencing parameters can e.g. be:

Constraints: The number of constraints, e.g. real-time constraints in the project plays a fundamental role as well their hardness influences how difficult they are to meet. Similarly the fulfillment of some constraints is easier to verify than others, which further influences on the effort needed.

Project size: An easy description of the size of a project is the number of individual components to be implemented. In general, the more components invloved, the longer the implementation time.

Input/output signals: The number of connections and signals between the internal components influence the design since every time a signal enters a component the component needs to act on it. Thus, more signals bring more parameters into the component which very often leads to an increased complexity.

Complexity: Describes in how many different ways the components in the algorithm can be concatenated. A complex algorithm where the components are intensively linked will in general take longer time to implement.

Novelty: If the components in the algorithm are well known it may not be necessary to re-implement them. E.g. if the designer is dealing with an application which has already been implemented, an IP (in-house or from a vendor) could be reused and potentially shorten the implementation time. Moreover, existing knowledge about the feasibility of the implementation will reduce the number of needed investigations of the algorithm to architecture mapping.

The third category is the architecture, e.g. the influencing parameters of the (HW/SW) platform where the algorithm is to be executed and of the supporting tools suites. A non-exhaustive list of these parameters is shown below:

Complexity of the architecture: The complexity of the architecture depends on the internal organisation and the features of the processing and communication elements. These include among others support for parallelism, VLIW capability, low level configurability, I/O features, pipelining and superscalar capability and branch prediction. Exploiting a complex architecture usually enables the designer to meet performance constraints more easily, at the expence of a longer development time.

Instruction support: The availability of specialised instructions or dedicated circuits which support the algorithm will usually enable the designer to arrive at a solution faster.

Debugging tools: Good debugging tools will usually result in a faster development time.

Synthesis/compile time: The architecture is supported by a set of tools. Not only the designer's ability to use the tools, but also the speed of the tools for synthesis and compilation, can have an influence on the development time, especially during the debugging phase.

When considering the above mentioned influencing parameters in the three categories we have listed (please note that the lists are not exhaustive), it is clear that there are many other influencing parameters which have an impact on the implementation effort. Hence it is therefore not straight forward to estimate the implementation effort. Considering some of the parameters, it is clear that they are not easily measurable. This is especially true for some of the human influencing parameters such as skills and social atmosphere, in addition to other parameters such as novelty. In the ideal case, data of all influencing parameters should be considered when estimating the implementation effort.

The task of estimating the implementation effort can be seen as a mapping of all the influencing parameter into a single variable. It would

be a very large task to obtain data for all influencing parameters, if at all possible. Moreover, not all parameters have the same significance. When devising a measure for estimating the implementation effort, it is important to at least include the influencing parameters which distinguish one implementation from another. As we are evaluating different algorithms to estimate their respective implementation times, the algorithm itself is a major parameter in the estimation process. It is therefore important to include a parameter which can express the algorithm. Eventhough, the parameters related to the designer are important, they can be rather constant. Of course the level of social interaction and cooperation variate from design to design and company to company, but they can be generalised and assumed to be constant.

When estimating the implementation effort with a limited subset of parameters, there will be factors which are not taken into account and thus limit the accuracy of the results. However, by using the principles of "other things being equal"[5] and thereby assuming that the impact of most parameters can be generalised to be a constant, we can see the impact of a few parameters. In Paper A, the algorithm and the designers' experience are the only variable parameters, and we see a large correlation between these and the needed implementation effort.

3.5 Research Thesis

We have now presented the background for this work, been though the typical design trajectory, and discussed performance and processes aspects of modern embedded system design.

Together with the observations of how design partitioning is conducted in real life in many small and medium sized companies, there is a gap between a first approach based on experience and intuition and a second one based on methods which aim to have a high fidelity towards the final implementation performance.

Furthermore in the current partitioning methods, focus is mostly on the performance of the final implementation. However, methodological aspects can be vital for the success of product development, and should be taken into account.

[5] In Latin: Ceteris paribus

Therefore the research thesis of this work is twofold: firstly it must be possible to partition the design based on metrics which are light, fast and easy to obtain and still able to provide sufficient fidelity towards the final implementation. This is to be contrasted to current methods which use dynamic and static metrics to obtain their high fidelity, and to the use of intuition and experience. Such an approach should be considered as a light variant of design space exploration, or DSE light. Secondly, the typical performance oriented cost function for guiding the partitioning can profitably be complemented with a metric which reflects the implementation effort. Such an addition provides valuable information to the designer for considering the time-to-market issue when partitioning the system.

4 Design Space Exploration - State-of-the-Art

In the previous sections we have considered the typical design trajectory for designing a heterogeneous multiprocessor system. We have identified the pre-analysis as a significant phase when dealing with the increased complexity of the system. In particular the HW/SW partitioning problem has a great influence on the solution's ability to meet the constraints from a given specification. Furthermore, we have discussed the different criteria which can be used for evaluating different solutions in order to select the right partitioning. The criteria result in a set of cost-parameters (Time, Area, Power, Implementation Effort, etc.) and we have demonstrated which possible diversity of influencing parameters influences such cost-parameters. We have also discussed the different classes of analysis which can be used to quantify the influencing parameters. In this section we give a non-exhaustive overview of the research field related to execution time estimation and implementation effort prediction.

4.1 Execution Time

The time, or more precisely the execution time of the algorithm, is in many cases the most dominant cost-parameter. For many applications, especially in the area of signal processing, the specifications contains information about how fast the algorithm(s) need to be executed in order to interact with other application(s) or users. The constraints in the speci-

fication will, in many cases, include a real time requirement (soft or hard real time), which needs to be met in order to achieve a satisfying solution.

The execution time of an algorithm on a given architecture is not always deterministic. It can be influenced by many run time factors such as:

- Data dependencies;
- Scheduler (OS Scheduler, hardware scheduler);
- Hazards (Cache misses, pipeline stalls, etc.);
- Out-of-order instruction execution.

Because of this non-determinism, different estimates of the execution time exist. Most estimation methods focus on estimating the worst case execution time (WCET) which usually refers to the longest execution path. However, there are designs where the best case execution time or the average case execution time are sufficient for partitioning the system.

In order to estimate the execution time, two models are required: one of the behaviour of the algorithm and one of the architecture. Based on these models, estimating the execution time involves three steps: Allocation, Scheduling and Mapping. The allocation step involves determining the needed number of processing elements for the execution. The scheduling step determines in which order the operations in the algorithm are executed and the mapping describes on which processing elements the operations are executed.

The accuracy of the estimates depends on many factors. Firstly, it depends on whether the data is based on a static analysis of the algorithm or on a set of representative data as input to a dynamic simulation. Secondly, it depends on the methods used to carry out the three above mentioned steps. And thirdly, it depends on the accuracy of the models involved.

In what follows we describe various strategies for estimating the execution time. As already mentioned, these can be classified into two analysis classes: Dynamic and Static, which are further explained here.

Dynamic Execution Time Estimation

For the dynamic class, a few different approaches for measuring the execution time are used. The dynamic analysis classes work by simulations whereby the behavioural model of the algorithm is executed on a model of the architecture. The model contains information about the time required for performing different operations. During the simulation the number of occurrences of the different operations are summed up (using more or less advanced models) into an execution time measure.

As mentioned earlier, the simulation can be performed on many different levels of abstraction. Independently from the abstraction level, different models of computation are used to represent the architecture. For example in case of system-level simulation, models of the architecture are often build by means of Kahn process networks [30], or similar variants of it. Another used model of computation to simulate the system are the Petri nets [31], or variants hereof. Finite state machines and an uncountable number of variants of if are another way of modelling.

When targeting more accurate simulation results, the models of the hardware become more detailed. Here we see a trend in moving away from formalised models of computation and just using the specific hardware modelling languages. Those languages include SystemC [32] and SpecC [33], which support different abstraction levels e.g. Transaction Level Modelling (TLM), Cycle accurate, etc. For a more thorough survey and more extensive explanation of different models of computation used for dynamic analysis, we recommend [34].

Static Execution Time Estimation

In this section we introduce some of the most relevant works about static execution time estimation relevant to the context of this thesis.

As opposed to the dynamic class, static estimation is not simulation based, i.e. it does not execute a model of the algorithm on a model of the architecture. Instead, static estimation bases its measures on collecting various pieces of information which are related to the execution time. This can be carried out on many levels using very simple architectural models. The different methods of collecting execution time related information fall into two categories: techniques estimating properties which indirectly report about the execution time, and techniques which directly relate

to an estimate of the execution time. A first example of the indirect methods is measuring the frequency of occurrence of a given operation in the execution path(s) extracted from a graph of the algorithm. A second example is to simply measure the frequency of occurrence of given operations. Examples of this direct method include measuring the length of the execution path(s) based on data-dependencies or simply measuring the number of executable lines of code.

Since the indirect methods do no give an estimate of the execution time, they are combined with the direct ones in order to emphasise a potential affinity between an algorithm and an architecture or to indicate possible overhead(s) that must be considered when evaluating the estimate of the execution time.

The direct methods include:

Graph based estimation: A very common way to statically estimate the execution time is to analyse the algorithm and construct a graph (E.g. CFG, CDFG, and DFG). From this graph, the longest execution path can be identified and used for estimating the execution time, which in many cases will correspond to the worst case execution time [35]. This approach is widely used, as for example in [36] and [37]. The execution time estimate can be expressed as:

$$\hat{t}_{exec} = \frac{\sum OperationsInExecutionPath}{\hat{f}_{arch}} \tag{1}$$

where $\sum OperationsInExecutionPath$ denote the number of operations in the longest execution path, and $\frac{1}{\hat{f}_{arch}}$ is an estimate of the execution speed of the architecture.

Source code based estimation: A less common, and simpler approach consists of counting the number of executable lines of source code or in cross-compiling the source code to assembly and performing a similar measure. This approach is seen in e.g. [38]. Such an approach does not take loops into account and is less accurate than the graph based approach.

Parallelism: γ, proposed by Le Moullec et al. [39], is a metric dealing with the potential speedup of an algorithm. The γ metric is calculated by measuring the ratio between the total number of operations

of the algorithm, $NoOp$ and the number of operations in the critical path, CP of the algorithm such as:

$$\gamma = \frac{NoOp}{CP} \tag{2}$$

The metric enables the designer to get an idea of the potential speedup of the algorithm when exploiting the parallelism following the ideas presented by Amdahl [40]. Exploiting the inherent parallelism of the algorithm on a supporting architecture directly influences the execution time.

Similar to γ, γ' proposed in Paper C, extracts the potential speedup of an algorithm but in a normalised way. It is defined as:

$$\gamma' = 1 - \frac{CP}{NoOp} \tag{3}$$

The γ' score is normalised and therefore it is easier to compare different algorithms or subparts hereof.

The indirect methods includes:

Memory Oriented Metric The memory oriented metric, MOM [39], is a metric measuring the ratio between the operations which have memory access and the total number of operations:

$$MOM = \frac{NoOp_{memoryAccess}}{NoOp} \tag{4}$$

The purpose of the MOM metric is to characterise the nature of the algorithm. Algorithms with a large number of memory access compared to the total number of operations will typically be very demanding on the access time and bandwidth to memory. A high score of MOM indicates that many memory operations occur which in turn can prolong the execution time because of e.g. slow external memories or cache misses.

Control Oriented Metric The control oriented metric, COM [39], is a metric measuring the ratio between the control operations (IF,

WHILE, FOR, SWITCH) and the total number of operations. The metric is calculated as:

$$MOM = \frac{NoOp_{Control}}{NoOp} \tag{5}$$

Just like MOM, the purpose of the COM metric is to characterise the nature of the algorithm. An algorithm with many control operations compared to the total number of operations will usually result in a large and complex control path when implemented on e.g. an FPGA. A high value of the COM metric indicates that close attention should be paid when addressing the control path in the design. Many steps in the control path can result in longer execution time. On platforms which use branch prediction and long pipelines this can result in penalties which also increase the execution time.

Closeness and Affinity metrics Metrics exploring the closeness or affinity between algorithms and architectures have been proposed by several researchers. Closeness metrics have been proposed in [41]. This set of normalised metrics has been devised for guiding the partitioning step at the system level. They are derived from a so-called Access Graph which is a directed graph that represents the access relations between behaviour and variables. Seven metrics are computed and used for clustering (and thus partitioning) purpose. The first three ones, connectivity, shared hardware, and sequential execution are used to characterize procedural-level behaviour: connectivity estimates the number of wires shared between two sets of behaviour, shared hardware measures the amount of hardware that two sets of behavior could share, and sequential execution is used to identify when two behaviors, defined sequentially in the specification, could be mapped to a single processing element in order to reduce communication overhead. The fourth one, communication, reflects the amount of data transferred during execution. The fifth one, constrained communication, considers both the communication and the provided performance constraints. The sixth one, common accessors, is an indirect measure used when the communication metric cannot be calculated. The seventh one, balanced size, is used to avoid too large clusters. By means of these closeness

metrics, it is possible to identify the most suitable utilisation of the platform and thus to minimise the execution time.

The affinity metric has been proposed by D. Sciuto et.al. in [42]. The objective of the affinity metric is to guide the design partitioning of the system specification between general purpose processors, DSP processor, and FPGA/ASIC. The metric consists of a triplet of values $(A_{GPP}, A_{DSP}, A_{FPGA})$ indicating the match between the processing elements and the examined code. The elements which compose the metric are derived from 14 other (sub)metrics which express the correlation between certain patterns in the code and the architectural properties. The submetrics are defined as ratios between lines with specific properties, e.g., the ratio between lines with a condition and the total number of lines, or defined as the number of assignment of a special type related to the total number of assignments. A high affinity score for one or several of the architectural categories indicates a high affinity between architecture and algorithm which should result in the ability to exploit architectural properties leading to a fast execution. In paper C we proposed an extension to the affinity metric which takes the parallelism into account and improves the accuracy of the measurement in relation to FPGA architectures.

Use of estimation methods

In the previous subsections we have discussed the principles of state-of-the-art estimation methods. In Table 1 we list a large part of frameworks, approaches, and methodologies which include dynamic and/or static estimation. It is not the purpose of this section to make a detailed description of the individual entries in the list; the interested reader can find extensive details about some of, but not all of, them in the surveys by Gerstlauer et al. [43] and Gries [13].

Looking at Table 1, it is clear that most work is done with a focus on dynamic estimation methods. It is also worth noting that many of the frameworks and methodologies which include static estimation use it as a preliminary partitioning or for reducing the design space before applying dynamic methods for obtaining more accurate estimates. This is for example the case for Koski [44], which is a UML based design flow

Estimation Method	Used in
Cycle Accurate	Disydent [45], MASIC [10], MetroPOLIS [7], Koski [44], CODESL [46], TOSCA: [47], MILAN [48]
Trace Based	MetroPOLIS [7]
Instruction Set Simulation	MetroPOLIS [7], CODESL [46]
Transaction Level Simulation	MetroPOLIS [7], Koski [44], MILAN [48], No Name [49]
System-Level Simulation	MetroPOLIS [7], Ptolemy II [50], PeaCE [51], SESAME [52] (now part of Daedalus [53]), No Name [54], No Name [55]
Synthesis based	Synforas' PICO Express [56]
Static Estimation	Koski [44], Affinity-Driven DSE: [38, 57], Design Trotter [37], CODESL: [46], iTuCoMe [58] No name [59], No Name [36]

Table 1: Overview of different estimation methods (with respect to execution time) and examples of frameworks, approaches, and methodologies where they are used.

for DSE on a high level of abstraction. The design space exploration is performed in two steps, using first a static analysis for a coarse grained pre-partitioning and then a dynamic analysis, where more cycle-accurate models of the processing elements are used for evaluation.

4.2 Measuring and Evaluating the Implementation Effort

In this section we present the concepts found in current methods used for quantifying the selected influencing factors as well as their corresponding metrics in the domain of implementation effort.

Measuring and estimating the implementation effort is a topic rooted in pure software development, mainly for large systems such as main-

frames. Most research has been carried out for pure software projects, but in the recent years embedded systems have begun to received a little more attention.

The core idea shared by all existing methods is twofold: firstly, some measurable properties of the algorithm are quantified, in order to form a size estimate of the algorithm. Secondly, the size estimate is fed into a model describing the relation between size and implementation time[6]. More concretely this can be expressed similar to the COCOMO 81 project (COnstructive COst MOdel) [60] and COCOMO II project [61] as $Effort = A * size^b$ where A and b are adjustable parameters which variates depending of the type of project, such as embedded systems or real-time systems, and should also reflect factors such as manpower and experience of the developers. In the COCOMO II project, 161 software development projects have been examined and proposals for the adjustable parameters (A and b) are described in [61]. A similar approach is the ISBSG method [62] which estimates the development time based on size of the problem, team size and type of project.

The size of a project can be measured in many ways. One of the first attempts to measure the size of a project was done by Albrecht [63] with the function point measure. Function points consist of two main stages: the first stage consists in counting and classifying the function types of the software: identified functions are weighted to reflect their complexity (Low, medium, high), usually based on the developers intuition and experience. This results in an unadjusted function point measure on which the ISO/IEC 20926:2003 standard is based. The second stage is the adjustment of the function points based on 14 parameters which are tuned according to the characteristics of the application and of its environment.

SPQR/20 (Software Productivity, Quality and Reliability with regard to 20 influencing factors) has been proposed by Jones [64] as a less heuristic-oriented variant of function point; experimental results [65] suggest that it can provide the same accuracy than function point while being simpler to work with.

A simplified function point measure has been developed by the Netherlands Software Metrics Users Association (NESMA) which only counts two of the five elements from the original function point [66].

[6] notice the difference with execution time

The function points are relatively easy to measure early in the development since they can work directly on the requirement specification. Another size measure which does not share this property is lines of code (LOC). On the other hand LOC is used extensively by the estimation techniques presented in the COCOMO projects. The function points can be converted into a LOC measure based on an implementation language-dependent factor. LOC has been criticized for its difficulty in handling different programming languages and productivity across different projects [67]. But according to [68] this is also the case for function points.

Similar to our earlier discussion regarding the parameters influencing the implementation effort in Section 3, the size of a software project in relation to the implementation time needed is influenced by multiple variables. Representing it in only one dimension brings uncertainty to the representation.

Although the cyclomatic complexity [69] metric was not originally intended for measuring the size of a project, Jay et al. [70] suggests that there exists a stable linear relationship between cyclomatic complexity and LOC. The cyclomatic complexity measures the number of linearly independent paths in an algorithm. As a difference to the function points measures, the cyclomatic complexity requires a detailed behavioural description to work on.

Besides the methods described above, which concentrate on size estimation of the project, an index based method is presented in Cadle and Yeates [71]. The method is called "analysis effort method" and its purpose is to estimate the needed effort for the analysis of a project. The method indexes three parameters: Size of the team, Familiarity, and Complexity. These parameters are rated from a predefined table, and once combined together, cross-checked with another table which describes the ratios between the needed effort for Analysis, Design, Coding and Unit Testing (CUT) and Testing. The method does not include any way to estimate the specific time needed for the individual phases but only the ratio.

A cost model which takes both the software and hardware aspects into account has been proposed by Agarwal and Shankar [72]. They introduce a cost model with the objective of understanding current Product Development Cycles (PDC) and evaluating the impact of new technologies on these PDC. In particular, the authors focus on cost and product develop-

ment time and propose a PDC model known as One Pass to Production (OPP). Although promising, their approach is very specific (they consider a FPGA-based NOC backbone) and the numerous assumptions made by the authors (e.g. regarding the number of required engineers) make the generalisation of their approach rather difficult.

To the best of our knowledge, very few works address the problem of estimating hardware implementation effort, and publications dealing with this topic are far less abundant than those dealing with software.

VHDL function point, presented in [73], elaborates on the function points analysis by modifying it to work with VHDL code. In this approach, the number of internal I/O signals and components is counted, and their counts are classified into levels. Subsequently, a function point value related to VHDL is extracted based on a predefined table. Experimental results considering the number of source lines in the LEON-1 processor project yields predictions which are within 20% of the real size.

An improvement to this can found in Paper A and in Technical Report B where the cyclomatic complexity [69] is used for estimating the size of a hardware project.

Other publications such as [74] compare actual hardware implementation efforts for different design methodologies but do not provide any systematic method to estimate the efforts.

5 Contributions and Conclusion

We have now presented the background of this work, been though the typical design trajectory, and discussed issues and problems related to modern design. Following this, we have formulated the research thesis of this work and have presented the state-of-the-art in design space exploration. We can now present the contributions of this work.

The papers which form the main body of this thesis fall into three different categories. Paper A and Tech Report B deal with the estimation of the implementation effort for implementing algorithms onto FPGAs. Paper C deals with extending a promising static partitioning approach with a parallelism measure to better handle the estimated performance of FPGAs. And finally, paper D shows some of the limitations of system level tools and static analysis.

Paper A: As we have seen through out this introduction, cost-parameters dealing with the design processes has not received nearly as much attention as the performance cost-parameters. In particular we have identified that implementation effort for hardware design has been neglected. In order to remedy this situation we decided to concentrate a significant part of this research on this highly applicable topic. This paper presents a metric-based approach for estimating the hardware implementation effort for an application in relation to the number of linear-independent paths of its algorithms. We adapt the cyclomatic complexity measure, and complement it with a function taking the designers' learning curve and experience into account. Our results show that a relation between the paths and the needed effort exists, and that it is thereby possible to estimate the hardware implementation effort.

Tech Report B: In this report we further investigate the issue of accurate implementation effort estimation with respect to real time constrained systems. We suggest a set of two metrics used to characterise the effects of implementation optimisations: one expressing how hard it is to reach an implementation satisfying the real-time constraints for the implementation, and another to reflect how the parallelism impacts optimisations. The Experimental results do not show an unambiguous result. However, for most algorithms our approach enables us together with the path measure from paper A to estimate the hardware implementation effort for hard real-time constrained applications.

Paper C: Approaches based on the gamma and affinity metrics are interesting for complementing other static analysis methods since they add information to time estimates and thus strengthen their value. However, the original gamma metric suffers from the fact that it is not normalized, which limits the extent with which it can be used to compare designs. Thus gamma does not reflect how parallelism is distributed i.e. how much of the algorithm can be parallelised. Furthermore, the original affinity metric does not take parallelism into account, thus it cannot reflect an algorithm's potential for FPGA implementation. In this paper we found that parallelism detection is essential when characterising an algorithm's potential for FPGA

implementation and therefore decided to improve the original affinity model so it includes parallelism. We show that it completely changes the FPGA affinity score for some algorithms.

Paper D: Static analysis is limited by the accuracy of the underlying models, especially when considering high level of abstractions such as system-level. In order to evaluate this type of limitation and to motivate the need for a metric based approach (cf. Paper C), a case study has been performed. In Paper D, which was the first paper we published, we show the differences between static estimation techniques and the performance of a system generated through system level tools. This was done by working with the implementation of turbo encoders and decoders for the HSDPA scheme for 3G networks.

As a wrap up of the individual conclusions we have hereby shown that the presented static metrics can provide a correlation between their estimates and reality. At the same time we consider these metrics to be easy to obtain and smoothly integrated as a parameter when partitioning a system. These results follows the initial research idea about improving current partitioning approaches based on experience and intuition by a light and fast analysis without performing more accurate but also more time consuming analysis. However, further investigation and extension are needed before the approach of a light design space exploration will become a self-contained methodology.

6 Glossary

In this glossary section, we define the essential terms which are used in this thesis. The glossary has been included as having many definitions in the main body of the thesis can be disruptive and hinder the smooth flow of ideas for the reader.

Abstraction Level: Abstraction level denotes the level of detail which is considered at a certain step in the design trajectory. A low level of abstraction means a more detailed and accurate description of the system.

Accuracy: Accuracy refers to the closeness of a measure or estimate of a quantity to its actual value. Accuracy is also used to characterize a measuring system or estimation method. Note that in scientific terms, accuracy is different from precision; see precision in this taxonomy.

Algorithm Transformation: Algorithmic transformations are techniques used to exhibit certain properties (such as parallelism) of an algorithm, without modifying the input-output relation, in order to improve its mapping on a given architecture.

Allocation: In hardware synthesis, allocation consists of determining the number of required resources (operators) for executing the operations of an algorithms. Allocation, binding, and scheduling influence each other.

Architecture: The architecture denotes both the target platform and its internal elements and structure.

Behaviour: The behaviour of a system is the description of what the system is supposed to output based on input stimuli. Strictly speaking, behaviour differs from functionality in terms of describing what the system does and not what it can do. In the literature they are often used interchangeably although authors use one or the other. Here we use them as synonyms.

Binding: In hardware synthesis, binding consists in deciding which operations should be executed by which operators. Binding, allocation, and scheduling influence each other.

Cycle Accurate: A simulator or model is said to be cycle accurate if it reflects the micro-architectural features (both HW/SW) of a platform with an accuracy corresponding to an instruction or a hardware cycle (i.e. the number of clock cycles required for the instruction or hardware cycle is known).

Execution Time: Execution time is the time it takes for executing the behaviour of an algorithm on a given platform.

Functionality: If a system is seen as a box, functionality describes what the box should output, possibly based on its input. In other words it describes which behaviour the system has, as defined by the designer. Strictly speaking functionality differs from behaviour in terms of describing what the system can do and not what it does, but in the literature they are often interchangeably. Here we consider them as perfect synonyms and use them as such.

Hardware/Software partitioning: Hardware/Software partitioning is the task of deciding onto which architectures the algorithms of an embedded system should be implemented.

Instruction Set Simulation: Instruction Set Simulation (ISS) refers to simulation conducted by means of an architectural model and which reflects the behaviour of a microprocessor with a precision corresponding to an instruction (usually with no information about the number of clock cycles per instruction). ISS is usually faster than cycle accurate simulation, but less accurate.

IP: In this thesis, IP (Intellectual Property) blocks refer to already existing cores for FPGAs and ASICs.

Mapping: The goal of the mapping process is to associate efficiently the functionality of the application to the target platform. It is a combined task consisting of allocation and scheduling of the operations to operators.

Operations: Operations are the computations (i.e. arithmetic as well as conditional) and memory accesses in the algorithm.

Operators: Operators are predefined architectural elements or combinations of architectural elements which can execute operations.

Platform: A platform is a set of processing elements interconnected in a predefined topology onto which the functionality can be mapped and executed. The platform can also include a service layer in terms of software such as an operation system.

Precision: Precision refers to the reproducibility or repeatability of a measurement system or estimation method. As such, a precise mea-

surement system or estimation method is not necessarily accurate; see accurate in this taxonomy.

Processing Element: At any abstraction levels, a processing element refers to a unit which is able to process data. E.g. it can be a computer in a network, a FPGA or DSP in a system, or at a lower level an operator inside a DSP or FPGA.

Real-Time: A real-time system is subject to real-time constraint(s), i.e. it must respond to events within a given deadline.

Scheduling: In hardware synthesis, scheduling consists in deciding in which order the operations of an algorithm should execute. Scheduling, binding, and allocation influence each other.

Simulation: Simulation is the activity consisting of the execution of one or several functional models of the application on a model of the architecture to mimic the behaviour of a system. This can be carried out at any abstraction levels and can be used to evaluate the correctness and performance of the system.

Synthesis: The synthesis task is responsible for transforming a behavioural description (e.g. C or Behavioural VHDL) into a dedicated hardware block. Although many articles as well as EDA tools consider the synthesis to include only the creation and instantiation of operators in logic gates, here we also include the place and route task.

System-Level Simulation: System-Level Simulation is conducted using a System Level Model, i.e. a model of the behaviour of the entire system described in a high-level language. System Level is often used to refer to levels above register transfer level.

Test: Tests refer to the experiments which are conducted in a systematic procedure to measure and evaluate certain properties of a system, often for verification purposes. See verification and validation entries in this list.

Trace Based: Trace based refers to methods and techniques which rely on the software and/or hardware traces, i.e. a recorded list of the execution steps taken by an algorithm onto a platform.

Transaction Level Simulation: Transaction Level Simulation is based on Transaction-Level Modeling (TLM), i.e. a high-level approach which model hardware/software systems by means of communicating modules (which often do not contain no or little details about the actual implementation). Communication between modules are modeled with channels, and are presented to modules by means of e.g. interface classes. Transaction Level Simulation is often faster but less accurate than e.g. ISS or cycle accurate simulations.

Validation: Validation is the process which consists in insuring that a system accomplishes its intended overall purpose. Often expressed by "Are you building the right thing?". See test and verification entries in this list.

Verification: Verification is the process which consists in checking by e.g. means of tests whether or not a system complies with the initial specifications. Often expressed by "Are you building it right?". See test and verification entries in this list.

VLSI technology: VLSI (Very Large Scale Integration) refers to the design and implementation of integrated circuits which include thousands to billions of transistors into a single chip.

7 List of Abbreviations

ASIC: Application Specific Integrated Circuit

CDFG: Control Data Flow Graph

CFG: Control Flow Graph

CPU: Central Processing Unit

DFG: Data Flow Graph

DSE: Design Space Exploration

DSP: Digital Signal Processor

FPGA: Field Programmable Gate Array

FSM: Finite State Machine

GPP: General Purpose Processor

HW: Hardware

IP: Intellectual Property

IPC: Inter Processor Communication

I/O: Input/Output

LOC: Lines Of Code

NRE: Non Recurring Engineering

OS: Operating System

RTL: Register Transfer Level

SDF: Synchronous Data Flow

SME: Small and Medium Enterprise

SW: Software

TLM: Transaction Level Modeling

UML: Unified Modeling Language

VHDL: VHSIC Hardware Description Language

VHSIC: Very High Speed Integrated Circuit

VLIW: Very Long Instruction Word

VLSI: Very Large Scale Integration

References

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, April 1965. [Online]. Available: http://dx.doi.org/10.1109/JPROC.1998.658762

[2] K. H. Pries and J. M. Quigley, *Project Management of Complex and Embedded Systems: Ensuring Product Integrity and Program Quality; electronic version.* Hoboken, NJ: Taylor & Francis Ltd, 2008.

[3] D. D. Gajski, J. Zhu, R. Döner, A. Gerstlauer, and Shuquing, *SpecC: Specification Language and Methodology.* Klüwer Academic Publishers, 2000, iSBN: 0-7923-7822-9.

[4] A. Sangiovanni-Vincentelli, "Quo vadis, sld? reasoning about the trends and challenges of system level design," *Proceedings of the IEEE*, vol. 95, no. 3, pp. 467–506, March 2007.

[5] G. Martin. (2007, May) Barriers to esl adoption. [Online]. Available: http://www.soccentral.com/results.asp?CatID=574&EntryID=22779

[6] A. Liu, J. Feng, and J. Hu, "Research on soc hardware/software co-design platform based on mda," in *Computer-Aided Industrial Design Conceptual Design, 2009. CAID CD 2009. IEEE 10th International Conference on*, nov. 2009, pp. 2105 –2109.

[7] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *Computer – IEEE Computer Society*, vol. 36, no. PART 4, pp. 45–52, 2003, iSSN 0018-9162.

[8] A. Jantsch, S. Kumar, and A. Hemani, "The rugby meta-model," Royal Inistitute of Technology, Royal Institute of Technology, Department of Electronics, Electronic System Design Laboratory, Electrum 229, S-164 40 Kista, Sweden, Tech. Rep., 2000.

[9] K. Gruttner, F. Oppenheimer, and W. Nebel, "Osss methodology - system-level design and synthesis of embedded hw/sw systems in

c++," in *Applied Sciences on Biomedical and Communication Technologies, 2008. ISABEL '08. First International Symposium on*, Oct 2008, pp. 1–5.

[10] A. Deb, A. Jantsch, and J. Oberg, "System design for dsp applications using the masic methodology," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 1, feb. 2004, pp. 630 – 635 Vol.1.

[11] "Catapult c," Mentor Graphics, Tech. Rep., 2010. [Online]. Available: http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/

[12] N. Nojiri and T. Ishii, "Seamless top-down flow for quick trial of hw/sw co-design," in *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*, dec. 2003, pp. 8 – 12.

[13] M. Gries, "Methods for evaluating and covering the design space during early design development," *Integration, the VLSI Journal*, vol. 38, no. 2, pp. 131 – 183, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/B6V1M-4CVR4RB-1/2/eccf89245270b7dcdb4f3eef3be3c260

[14] J. H. Holland, *Adaptation in Natural and Artificial Systems*, 2nd ed. MIT Press, 1992, first edition published 1975.

[15] R. P. Dick and N. K. Jha, "Mogac: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 17, pp. 920–930, 1998.

[16] G. Ascia, V. Catania, and M. Palesi, "A ga-based design space exploration framework for parameterized system-on-a-chip platforms," *Evolutionary Computation, IEEE Transactions on*, vol. 8, no. 4, pp. 329 – 346, aug. 2004.

[17] T. Blickle, J. Teich, and L. Thiele, "System-level synthesis using evolutionary algorithms," *Design Automation for Embedded Systems*, vol. 3, no. 1, pp. 23–58, 1998. [Online]. Available: http://www.springerlink.com/content/l107341n77048127

[18] Y. Liu and Q. C. Li, "Hardware software partitioning using immune algorithm based on pareto," in *International Conference on Artificial Intelligence and Computational Intelligence*, 2009, pp. 176–180.

[19] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983. [Online]. Available: http://www.jstor.org/stable/1690046

[20] V. Cerny, "A thermodynamical approach to the travelling salesman problem: an efficient simulation algorithm," *Journal of Optimization Theory and Applications*, vol. 45, pp. 41–51, 1985.

[21] S. Banerjee and N. Dutt, "Efficient search space exploration for hw-sw partitioning," in *Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on*, sept. 2004, pp. 122 – 127.

[22] J. Olson, J. Rozenblit, C. Talarico, and W. Jacak, "Hardware/software partitioning using bayesian belief networks," *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, vol. 37, no. 5, pp. 655 – 668, Sept. 2007. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSMCA.2007.902623

[23] A. Sameh and W. Raslan, "Hardware/software co-design using bayesian belief networks," in *Proceedings of the 5th WSEAS International Conference on Signal Processing*, Istanbul, Turkey,, May 2006, pp. 82–87.

[24] R. Kennedy, J.; Eberhart, "Particle swarm optimization," in *Proceedings of IEEE International Conference on Neural Network*, 1995, pp. 1942–1948.

[25] M. Abdelhalim and S.-D. Habib, "Modeling communication cost and hardware alternatives in pso based hw/sw partitioning," in *Microelectronics, 2007. ICM 2007. Internatonal Conference on*, dec. 2007, pp. 111 –114.

[26] Z. A. Mann and A. Orbán, "Optimization problems in system-level synthesis," in *Proceedings of the 3rd Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications*, 2003. [Online]. Available: citeseer.ist.psu.edu/mann03optimization.html

[27] W. Jigang, T. Srikanthan, and G. Chen, "Algorithmic aspects of hardware/software partitioning: 1d search algorithms," *IEEE TRANSACTIONS ON COMPUTERS*, vol. 59, no. 4, pp. 532–544, April 2010.

[28] T. S. Olesen, "Private conversation about Danish SMEs design methodologies," August 2006.

[29] S. McConnell, *Software Estimation – Demystifying the Black Art*. Wasthington: Microsoft Press, 2006.

[30] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of IFIP Congress 74*, J. L. Rosenfeld, Ed. Stockholm, Sweden: North-Holland Publishing Co., August 1974.

[31] C. A. Petri, "Kommunikation mit automaten," Ph.D. dissertation, University of Bonn, 1962.

[32] (2010) System c language. [Online]. Available: http://www.systemc.org

[33] L. Cai, D. Gajski, and M. Olivarez, "Introduction of system level architecture exploration using the specc methodology," in *ISCAS*, vol. 5, 2001, pp. 9–12.

[34] E. A. Lee, "Embedded software," *Advances in Computers*, vol. 56, pp. 56–97, 2002.

[35] P. Altenbernd, "On the false path problem in hard real-time programs," in *8th Euromicro Workshop on Real-Time Systems*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 1996, p. 0102.

[36] M. Holzer and M. Rupp, "Static estimation of execution times for hardware accelerators in system-on-chips," in *System-on-Chip, 2005. Proceedings. 2005 International Symposium on*, nov. 2005, pp. 62 –65.

[37] Y. Le Moullec, J.-P. Diguet, N. B. Amor, T. Gourdeaux, and J.-L. Philippe, "Algorithmic-level specification and characterization of embedded multimedia applications with design trotter," *Journal of VLSI signal Processing*, vol. 42, pp. 185–208, 2006.

[38] L. Pomante, "System-level co-design of heterogeneous multiprocessor embedded systems," Ph.D. dissertation, Politecnico di Milano, 2002.

[39] Y. Le Moullec, N. B. Amor, J.-P. Diguet, M. Abid, and J.-L. Philippe, "Multi-granularity metrics for the era of strongly personalized SOCs," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2003.

[40] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS spring joint computer conference*, 1967.

[41] F. Vahid and D. Gajski, "Closeness metrics for system-level functional partitioning," in *In Proceedings of the European Design Automation Conference (EuroDAC)*, 1995.

[42] D. Sciuto, F. Salice, L. Pomante, and W. Fornaciari, "Metrics for design space exploration of heterogeneous multiprocessor embedded systems," in *Hardware/Software Codesign, 2002. CODES 2002. Proceedings of the Tenth International Symposium on*, 6-8 May 2002, pp. 55–60.

[43] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich, "Electronic system-level synthesis methodologies," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 28, no. 10, pp. 1517–1530, 2009.

[44] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, J. Riihimäki, and K. Kuusilinna, "Uml-based multiprocessor soc design framework," *ACM Trans. Embed. Comput. Syst.*, vol. 5, no. 2, pp. 281–320, 2006.

[45] I. Auge, F. Petrot, F. Donnet, and P. Gomez, "Platform-based design from parallel c specifications," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, no. 12, pp. 1811 – 1826, dec. 2005.

[46] Y. Hau, M. Khalil-Hani, and M. Marsono, "Codesl: A framework for system-level modelling, co-simulation and design-space exploration

of embedded systems based on system-on-chip," in *Intelligent Systems, Modelling and Simulation (ISMS), 2010 International Conference on*, jan. 2010, pp. 122 –127.

[47] A. Balboni, W. Fornaciari, and D. Sciuto, "Partitioning and exploration strategies in the tosca co-design flow," in *Hardware/Software Co-Design, 1996. (Codes/CASHE '96), Proceedings., Fourth International Workshop on*, 18-20 March 1996, pp. 62–69.

[48] V. Mathur and V. Prasanna, "A hierarchical simulation framework for application development on system-on-chip architectures," in *14th Annual IEEE International ASIC/SOC Conference*, 2001, pp. 428–434.

[49] I. Viskic, L. Yu, and D. Gajski, "Design exploration and automatic generation of mpsoc platform tlms from kahn process network applications," *SIGPLAN Not.*, vol. 45, no. 4, pp. 77–84, 2010.

[50] E. A. Lee, "Overview of the ptolemy project," University of California, Berkeley, CA, 94720, USA,, Technical Memorandum UCB/ERL M03/25, July 2003. [Online]. Available: http://ptolemy.eecs.berkeley.edu/publications/papers/03/overview/overview03.pdf

[51] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo, "Peace: A hardware-software codesign environment for multimedia embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 3, pp. 1–25, 2007.

[52] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transaction on Computers*, vol. 55, no. 2, pp. 1–14, February 2006.

[53] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere, "Daedalus: toward composable multimedia mp-soc design," in *DAC '08: Proceedings of the 45th annual Design Automation Conference*. New York, NY, USA: ACM, 2008, pp. 574–579.

[54] A. Bouchhima, P. Gerin, and F. Petrot, "Automatic instrumentation of embedded software for high level hardware/software cosimulation," in *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, jan. 2009, pp. 546 –551.

[55] L. P. M. Benders, "Specification and performance analysis of embedded systems with coloured petri nets," *Computers & Mathematics with Applications*, vol. 37, no. 11-12, pp. 177–190, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/B6TYJ-3WWM078-S/2/c01e93a1c8ba773c106812ecb1d6d7f2

[56] "Synfora pico," 2010, http://www.Synfora.com.

[57] C. Brandolese, W. Fornaciari, L. Pomante, F. Salice, and D. Sciuto, "Affinity-driven system design exploration for heterogeneous multiprocessor soc," *Computers, IEEE Transactions on*, vol. 55, no. 5, pp. 508–519, May 2006.

[58] H. Wang, J. Bian, Q. Wu, and Y. Wang, "itucome: Hcdfg-based incremental tuning hw/sw co-design methodology for multi-level exploration," in *Computer Supported Cooperative Work in Design, 2005. Proceedings of the Ninth International Conference on*, vol. 2, may 2005, pp. 978 – 983 Vol. 2.

[59] F. Graziosi, L. Imbriglio, and L. Pomante, "Topology-oriented system design exploration for embedded applications implemented onto heterogeneous multiprocessor soc," in *International Conference on Design & Technology of Integrated Systems in Nanoscale Era*, 2008, pp. 1–6.

[60] B. W. Boehm, *Software Engineering Economics.* Prentice Hall, 1981.

[61] B. W. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece, *Software Cost Estimation with COCOMO II.* Prentice Hall, 2000.

[62] P. Hill, *Practical Project Estimation*, 2nd ed. International Software Benchmarking Standards Group, 2005.

[63] A. J. Albrecht, "Measuring application development productivity," in *Proc. IBM Applications Development Symp.*, 1979.

[64] T. Jones, *Programming Productivity*. New York: McGraw–Hill, 1986.

[65] D. Jeffery, G. Low, and M. Barnes, "A comparison of function point counting techniques," *Software Engineering, IEEE Transactions on*, vol. 19, no. 5, pp. 529–532, May 1993.

[66] R. D. Stutzke, *Estimating Software-Intensive Systems*. Addison-Wesley, 2005.

[67] C. Jones, *Applied Software Measurement: Assuring Productivity and Quality*, 2nd ed. Mcgraw-Hill, 1997.

[68] L. H. Putnam and W. Myers, *Five Core Metrics: The Intelligence Behind Successful Software Management*. Dorset House Publishing Company, 2003.

[69] T. J. McCabe, "A complexity measure," *IEEE Transaction on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, December 1976.

[70] G. Jay, J. E. Hale, R. K. Smith, D. Hale, N. A. Kraft, and C. Ward, "Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship," *Journal of Software Engineering and Applications*, vol. 2, no. 3, pp. 137–143, Oct 2009. [Online]. Available: http://www.scirp.org/Journal/PaperDownload.aspx?paperID=779&fileName=JSEA20090300001_74742661.pdf

[71] J. Cadle and D. Yeates, *Project Management for Information Systems*, 5th ed. Prentice Hall, December 2007.

[72] A. Agarwal and R. Shankar, "Cost feasibility analysis for embedded system development and the impact of various methodologies on product development cycle," Florida Atlantic University, Tech. Rep., 2008.

[73] W. Fornaciari, F. Salice, U. Bondi, and E. Magini, "Development cost and size estimation starting from high-level specifications," in *Proceedings of the ninth international symposium on Hardware/software codesign*, 2001, pp. 86–91.

[74] L. Piga and S. Rigo, "Comparing rtl and high-level synthesis methodologies in the design of a theora video decoder ip core," in *Programmable Logic, 2009. SPL. 5th Southern Conference on*, 2009.

Paper A

A Priori Implementation Effort Estimation for HW Design Based on Independent-Path Analysis

Rasmus Abildgren, Jean-Philippe Diguet, Pierre Bomel, Guy Gogniat, Peter Koch, and Yannick Le Moullec

This paper has been published in
EURASIP Journal on Embedded Systems,
vol. 2008, pp.1–12, September 2008.

The layout has been revised.

Abstract

This paper presents a metric-based approach for estimating the hardware implementation effort (in terms of time) for an application in relation to the number of linear independent paths of its algorithms. We exploit the relation between the number of edges and linear independent paths in an algorithm and the corresponding implementation effort. We propose an adaptation of the concept of cyclomatic complexity, complemented with a correction function to take designers' learning curve and experience into account. Our experimental results, composed of a training and a validation phase, show that with the proposed approach it is possible to estimate the hardware implementation effort. This approach, part of our light design space exploration concept, is implemented in our framework 'Design-Trotter' and offers a new type of tool that can help designers and managers to reduce the time-to-market factor by better estimating the required implementation effort.

1 Introduction

1.1 Discussion of the Problem

Companies developing embedded systems based on high-end technology in areas such as telecommunication, defence, consumer products, healthcare equipment are evolving in an extremely competitive globalised market. In order to preserve their competitiveness, they have to deal with several contradicting objectives: on the one hand they have to face the ever-increasing need for shorter time-to-market and on the other hand they have to develop and produce low-cost, high-quality and innovative products.

This raises major challenges for most companies, especially for small and medium-sized enterprises (SMEs). Although SMEs are under pressure due to the above-mentioned factors, they are either not applying the latest design methodologies or can not afford the modern Electronic System Level (ESL) design tools. By limiting themselves to traditional design methodologies, SMEs make themselves more vulnerable to unforeseen problems in the development process, making the time-to-market factor one of the most critical challenges they have to deal with. A survey

released at the Embedded Systems Conference (ESC 2006) [1] indicated that more than 50% of embedded design projects are running behind schedule (i.e., 25% are 1-2 months late, 18% 3-6 months). In the 2008 version of the survey [2] it is again shown that meeting the schedule is the greatest concern for design teams.

Moreover, a workshop [3] held for Danish SMEs working in the domain of embedded systems clearly indicates that there is a need for changing and improving their design trajectories in order to stay in front on the global market. More specifically, this calls for setting modern design, i.e., hardware/software (HW/SW) co-design and ESL design into actual practice in SMEs, so that they can reduce their time-to-market factor and keep up with their competitors by being more efficient in producing embedded systems.

Although HW/SW co-design and ESL design tools (both commercial and academic) have been available for several years, there are several barriers that, so far, have prevented their wide adoption. To name a few:

- Difficulty in transfering the methods and tools developed by academia into industry, because they are mostly developed for experimenting with, validating and proving new concepts rather than for being used in companies. Therefore adapting and transferring these methods and tools require additional and tedious efforts, delaying their adoption,

- financial cost in terms of tool licenses, training, etc. that many SMEs cannot afford, since the cost of a complete commercial tool chain can exceeds in excess of 150 k per year,

- training cost and knowledge management issues, meaning switching to a new design trajectory also involves the risk of loosing momentum, i.e., loosing time and efficiency because of the training needed to master the new methods and tools,

- finally, many modern design flows are not mature enough to generate efficient and automatic real-time code, and combined with the previous item, cause potential adopters to wait until it is safe to switch.

Considerable research has been undertaken to estimate implementation factors such as: area, power, and speed-up that are subsequently used in HW/SW partitioning tools with different focuses related to granularity, architecture model and communication topology, and so on. All of these research projects does not include the man-power cost which is the most critical one for many companies, and especially SMEs. This work takes its outset in a research framework facilitating the HW/SW partitioning step for SMEs. It focuses on a light design space exploration approach called "DSE-light" that combines the advances in terms of design methodologies found in academia and the ease of integration required by SMEs, i.e., lowering the above-mentioned barriers.

The contribution presented in this paper is the development of a method for estimating the man-power cost (i.e., development time) for implementing hardware components and the integration of this method into our framework, so that HW/SW partitioning decisions can be wiser. A method that used iteratively and systematic will form the engine for precise development schedules. The following subsections present the rationale for this work and the idea enabling this contribution.

1.2 Parameters that Influence the Implementation Effort

A common problem in both SMEs and larger companies is that of estimating the amount of time required to map and implement an algorithm onto an architecture given parameters such as [4], [5]:

- manpower, i.e. the available development team(s) and their size(s),
- quality of the social interactions between the team members and the teams,
- experience of the developers (e.g. years of experience, previously developed projects, novelty of the current project, etc.),
- skills of the developers, i.e. their ability to solve problems (this is not the same as experience, which only reflects how often one has tried before),
- availability of suitable and efficient tools and how easy they are to learn and use,

- availability of SW/HW IP code/cores,
- involvement of the designers, i.e. are they working on other projects simultaneously?
- design constraints i.e. real-time requirements,

This work addresses the issue of adding man-power cost parameter into the cost-function and thereby guiding the HW/SW partitioning. More specifically we concentrate on the mapping process, i.e. the process of mapping a given algorithm onto a given architecture and the implementation effort (i.e. time) related to the complexity of that algorithm. Our framework also addresses other issues of HW/SW partitioning e.g. [6].

1.3 Idea

In order to understand what makes an algorithm difficult to implement, five semi-structured interviews[1] have been conducted with engineers (hardware developers) with very little to 20 years of experience.

From the interviews it was deduced that several parameters influence on the hardware design difficulty. The hardware developers stated that available knowledge about worst cases, dependencies between variables, and the completeness of the design description of the entire system including all communications are important for the design time. However, according to them, the major parameter influencing a hardware design is the number of connections and signals between the internal components. This should be viewed in the way that every time a signal enters a component, it means that the component needs to act on it. More signals bring more parameters into the component and that very often leads to an increased complexity.

Based on the interviews we form our hypothesis; it is that a strong relation exists between what renders an algorithm complex to implement and the number of components as well as the number of signals/paths in the algorithm.

To ensure that not only the number of paths are counted but also that a high number of components is present, we choose to only measure

[1]Semi-structured interview is an information-gathering method of qualitative research. Semi-structured interview is an adequate tool to capture how a person thinks of a particular domain [7].

the number of linear independent paths. Furthermore, this insures that components occuring several times during the execution are counted only once, which better reflects the actual implementation efforts.

The remainder of the paper is organised as follows: section 2 gives an overview of the state-of-the-art methods for estimating the implementation effort both for software and hardware designs and indicates the need for further work for hardware design. In section 3 a new metric for estimating the development time is defined and combined with our research tool "Design-Trotter". Section 4 presents some test cases used to investigate the validity of the above mentioned hypothesis and of the proposed metric. Furthermore, the experimental results are analysed. Finally we conclude in section 5.

2 State of the art

2.1 Software

Most research about estimating implementation effort is found in the software domain, especially within the COCOMO project [8]. The problem of estimating the implementation effort is twofold. First, a reasonable measure needs to be developed for being able to quantify the algorithm. Second, a model needs to be developed, describing a rational relation between the measure and the implementation effort.

COCOMO

To start with the model, a typical power model has been proposed inside the COCOMO experiment [8, 9]:

$$Effort = A \times Size^b \tag{A.1}$$

where $Size$ is an estimate of the project size, and A and b are adjustable parameters. These parameters are influenced by many external factors which we previously discussed in section 1.2, but can be trained, based on previous project data.

To use this COCOMO measure there is a need for expressing the size of the project. Inside the software domain the dominating metric is lines of code, LOC. Using LOC is not without difficulties, e.g. how is a code

line defined? [10] discusses this issue and states that LOC is not consistent enough for that use; this is also supported by [11]. Using the LOC metric also has several difficulties, e.g. it is not a language independent metric. Furthermore, hardware developers also tend to disapprove this measure, since they do not feel that it is a representative measure for hardware designs.

However, we do not claim that there is no relation between LOC and the implementation effort. It is impossible to write 10k lines in one day, but for VHDL the relation is not always straightforward. In the experiments that we have performed (data shown in Table A.1) there is no unambiguous relation between the LOC in VHDL and the development time.

[11] describes that making 'a priori' determination of the size of a software project is difficult especially when using the traditional lines of code measure; instead function points based estimation seems to be more robust.

Function Points Analysis

The function points metric was first introduced by Albrecht [12] and consists of two main stages. 1) Counting and classifying the function types for the software. The identified functions need to be weighted reflecting their complexity, that is determined on the basis of the developers' perception. 2) Adjustment of the function points according to the application and environment, based on 14 parameters. The function points can then be converted into a LOC measure, based on an implementation language dependent factor, and e.g. [11] reports that the function points metric can be used as implementation effort estimation metric. The function points analysis has been criticised of being too heuristic and [10] has proposed the SPQR/20 function points metric as an alternative. [13] has compared the SPQR/20 and the Function Points analysis and found their accuracy comparable even though the SPQR/20 metric is simpler to estimate.

2.2 VHDL Function Points

To the knowledge of the authors, limited research has been carried out in the field of estimating the implementation difficulty of hardware designs.

Fornaciari et al. [14] have taken up the idea from the function points analysis and modified it to fit VHDL. By counting the number of internal I/O signals and components, and classifying these counts into levels, they extract a function point value related to VHDL. They have related their measure to the number of source lines in the LEON-1 processor project and their predictions are within 20% of the real size. However, as stated previously, estimating the size does not always give an accurate indication of the implementation difficulty, and the necessary implementation time.

By measuring the number of internal I/O signals and components, their work goes along the same road as our initial observations indicate. However, our approach is pointing towards estimating the implementation effort, based on a behavioural description of the algorithm in the C-language. Furthermore, it also takes the designer's experience into account.

3 Methodology

The proposed flow for estimating the implementation effort is illustrated in Fig A.1. It takes its outset in a behavioural description of the algorithm, in C-language (including library function source code), which is intended to be implemented in hardware. From this description, we use the Design-Trotter Framework to generate a Hierarchical Control Data Flow Graph (HCDFG) which is then measured to identify the number of independent paths. The resulting measure, combined with the experience of the developers, gives an estimate of the required implementation effort. The method is self-learning in the sense that after each successful implementation, new knowledge about the developers involved can be integrated, and improve the accuracy of the estimates. The HCDFG and the approach for modelling the developers experience are covered later in this section but initially we investigate how the number of paths can be measured.

3.1 Cyclomatic Complexity

As described in section 1.3, the number of independent paths is expected to correlate with the complexity that the engineers are facing when working on the implementation. Therefore, finding a method to measure the

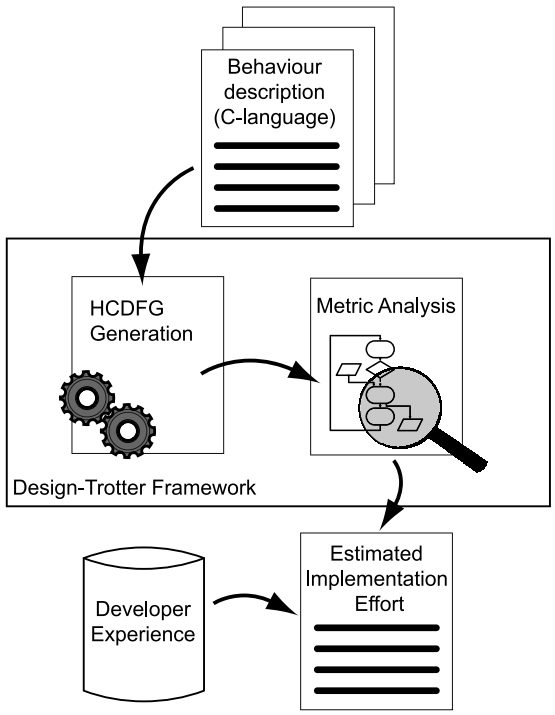


Fig. A.1: The flow of estimating the required implementation effort. The starting point is a behavioural description in C of the algorithm to be implemented in hardware (e.g. via VHDL). From this description, a HCDFG is generated and measured to identify the number of independent paths in the algorithm. This measure, combined with the experience of the developers, gives an estimate of the required implementation effort (expressed in time).

number of independent paths in an algorithm could help us investigating this issue. A metric measuring that is the cyclomatic complexity measure proposed by Thomas J. McCabe [15] which measures the number of linear independent paths in the algorithm.

The cyclomatic complexity was originally invented as a way to intuitively quantify the complexity of algorithms, but has later found use for other purposes especially in the software domain. The cyclomatic complexity has been used for evaluating the quality of code in companies [16], where quality covers aspects from understandability over testability to maintainability. It has also been shown [17] that algorithms

with a high cyclomatic complexity more frequently have errors than algorithms with lower cyclomatic complexity. The cyclomatic complexity has furthermore been used for evaluating programming languages for parallel computing [18], where languages that encapsulate control statement in instructions are receiving higher scores. All use the cyclomatic complexity measure under the assumptions that the complexity has significant influence on the number of paths the developers need to inspect, its correlation to the number of paths that needs to be tested, or a combination of the two.

In the domain of hardware, the cyclomatic complexity has also found use, judging the readability and maintainability in the SAVE project [19]. It is worth noticing that they use a misinterpreted [20] definition of the cyclomatic complexity [21].

All these projects utilise the cyclomatic complexity's ability to measure the number of independent paths and relate them to their individual cases.

The cyclomatic complexity is originally defined as a graph examination, however, the metric can be simplified [15] to:

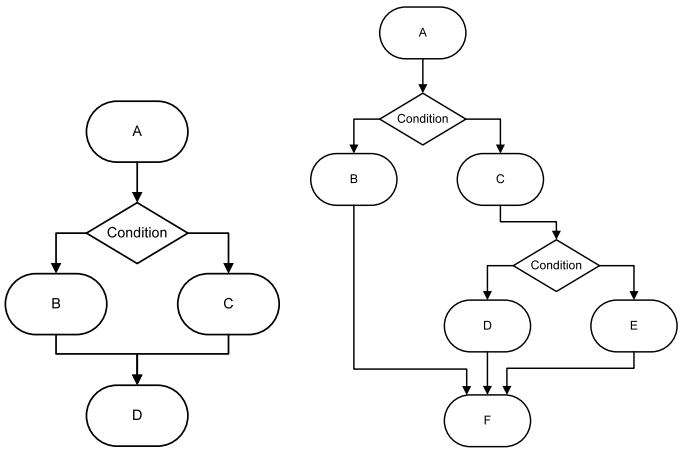
$$P(G) = \pi + 1 \tag{A.2}$$

where π represents the number of condition nodes in the graph G representing the algorithm being analysed. Fig. A.2 shows two examples of graphs and the corresponding cyclomatic complexity.

In this work we propose an adapted version of the cyclomatic complexity definition to estimate, a priori, the number of independent paths on a Hierarchical Control/Data Flow Graph (HCDFG), defined in the following section. The cyclomatic complexity for an HCDFG is obtained by examining its subgraphs as explained in section 3.3.

3.2 HCDFG

For this work we use the Hierarchical Control/Data Flow Graphs (HCDFG), which are introduced in [22] and [23]. The HCDFGs are used to represent an algorithm with a graph-based model so the examination task of the algorithm is eased. Control/Data Flow Graphs (CDFGs) are well accepted by designers as a representation of an algorithm where data flow graphs represent the data flow between different processes/operations,



(a) $P(G) = 2$ (b) $P(G) = 3$

Fig. A.2: Two examples of graphs for which the cyclomatic complexities have been calculated.

and the control flow layer, encapsulating these data flows and adding control structures to the graphical notation. The hierarchy layered structure is added to help representing large algorithms as well as to enable the analysis mechanism to identify functions/blocks in the graph. Such an identified block can then be seen as a single HCDFG that can be instantiated several times. Fig. A.3 shows an example of a Hierarchical Control/Data Flow Graph.

In this work the design space exploration tool 'Design-Trotter' is used as an engine for analysing the algorithms. The HCDFG model is used as 'Design-Trotter's' internal representation.

The hierarchy of an HCDFG is shown in Fig. A.3. An HCDFG can consist of other HCDFGs, Control/Data flow graphs (CDFGs) and data flow graphs (DFGs) as well as elementary nodes (processing, memory, and control nodes). An HCDFG is connected via dependency edges. In this work we only explore the graph at levels above the DFGs, and therefore only concentrate on these when we define the graph types in what follows.

Let us consider the Hierarchical Control Data Flow Graph, $G_{HCDFG} = (N_{HCDFG}, E_{HCDFG})$ where N_{HCDFG} are the nodes denoted by $N_{HCDFG} = \{n_{HCDFG_1}, \ldots, n_{HCDFG_m}\}$ and the nodes are

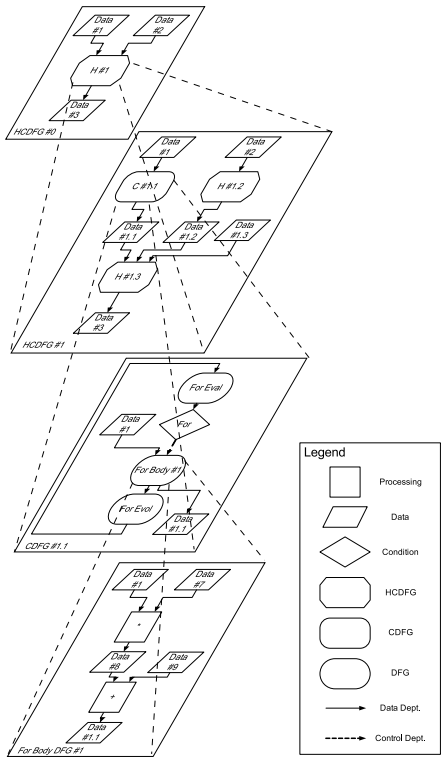


Fig. A.3: An overview of how the hierarchy in an HCDFG allows analysis of an algorithm on different levels and how the levels are related.

$N_{HCDFG} \in \{G_{HCDFG}|G_{CDFG}|G_{DFG}|Data\}$, meaning that the nodes in the G_{HCDFG} can be instances of its own type, encapsulated Control/Data Flow Graphs, G_{CDFG}, encapsulated Data Flow Graphs G_{DFG}, or data transfer nodes, $Data$. The last one is introduced to avoid the duplication of data representations in the hierarchy, when data is exchanged between the graphs. Thereby, data are only represented by their nodes and not by edges as it is common in many other types of DFGs.

The edges, E_{HCDFG}, connect the nodes such that $E_{HCDFG} = \{e_{n_{HCDFG_i},n_{HCDFG_j}}\}$ where $i \neq j$ and represent the indexes of the nodes, $E_{HCDFG} \in \{DD\}$ and where every node can have multiple input and/or output edges. For the G_{HCDFG}, only data dependencies, DD, are al-

lowed, and no control dependencies, CD.

In this way the HCDFG forms a hierarchy of encapsulated HCDFGs, CDFGs, and DFGs, connected via exchanging data nodes. The HCDFG can be seen as a container graph for other graphs types such as the CDFG.

We can define the CDFG as $G_{CDFG} = (N_{CDFG}, E_{CDFG})$ where N_{CDFG} are the nodes denoted by $N_{CDFG} = \{n_{CDFG_1}, \ldots, n_{CDFG_m}\}$ and the nodes are $N_{CDFG} \in \{CC|G_{HCDFG}|G_{DFG}|Data\}$ where $CC \in \{if|switch|for|while|do-while\}$. In this way the G_{CDFG} is able to describe common control structures, where the actual data processing is encapsulated in either DFGs or HCDFGs. Again, the data exchange nodes are used to exchange data between the other nodes.

The edges, E_{CDFG} , connect the nodes such that $E_{CDFG} = \{e_{n_{CDFG_i}, n_{CDFG_j}}\}$ where $i \neq j$ and represent the indexes of the nodes. If $n_{CDFG_i} \in CC$ and $n_{CDFG_j} \in \{G_{HCDFG}|G_{DFG}\}$ then $\{e_{n_{CDFG_i}, n_{CDFG_j}}\} \in \{CD\}$, else $\{e_{n_{CDFG_i}, n_{CDFG_j}}\} \in \{DD\}$

Beneath the Control/Data Flow Graphs G_{CDFG} the Data Flow Graphs G_{DFG} exist but they are of no use in this work so we will not define them further here.

3.3 Calculating the Cyclomatic Complexity on CDFGs

Now that the HCDFG has been defined, we explain our proposed method for measuring the cyclomatic complexity on the CDFGs.

Since the cyclomatic complexity only considers the control structure in finding the number of independent paths in the algorithm, the DFG part of the algorithm is, as mentioned earlier, of no interest for this task because it only gives a single path. On the other hand, what is of interest is how the cyclomatic complexity is measured on the CDFGs and HCDFGs which are built by the tool Design-Trotter. This leaves us with the following cases which are described in detail afterwards:

- If constructs
- Switch constructs
- For loops
- While/ Do-while loops

- Functions
- HCDFGs in parallel
- HCDFGs in serial sequence

If constructs

If constructs are represented as CDFGs, G_{CDFG}, where one node is a control node of type if (see Fig A.4(a)). Before arriving at the control node, a condition evaluation node $n_{eval} \in \{G_{HCDFG}|G_{DFG}\}$ is traversed to calculate the boolean variable stored in n_{Data} (to maintain simplicity these are not shown in Fig A.4(a)) that is used in the condition node. If the variable is true, the algorithm follows the path through the true body node, $n_{true} \in \{G_{HCDFG}|G_{DFG}|\emptyset\}$. Else it goes to the false body node $n_{false} \in \{G_{HCDFG}|G_{DFG}|\emptyset\}$. Note that in some cases, either the true body or the false body does not exist, but it still gives a path. In this case, according to the cyclomatic complexity measure, the number of independent paths is:

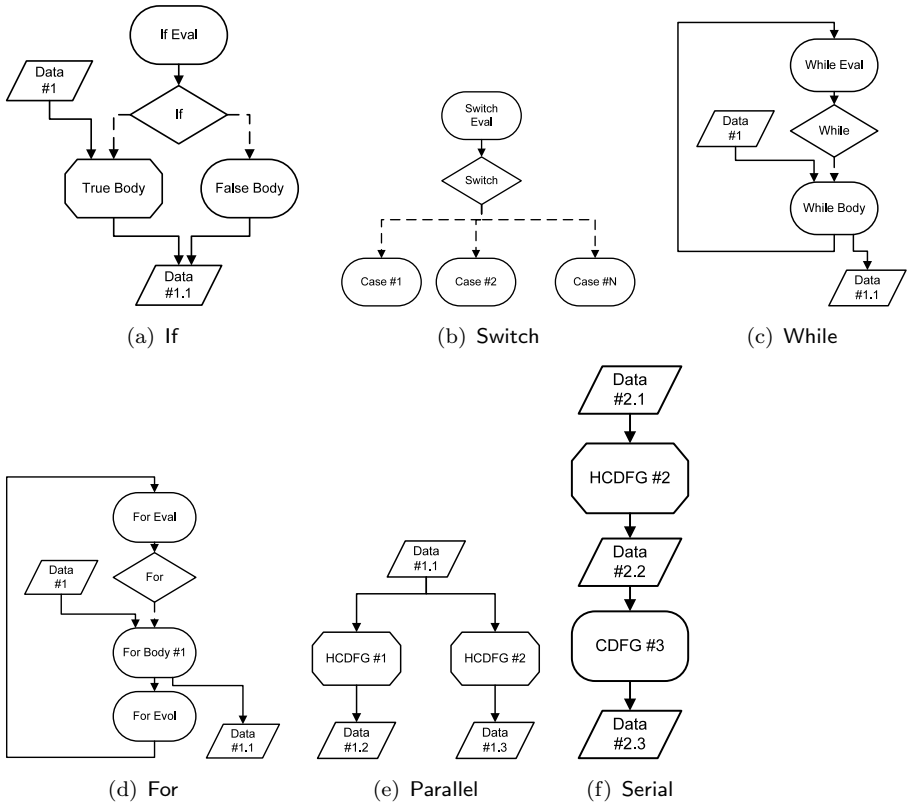
$$P(n_{if}) = P(n_{true}) + P(n_{false}) + P(n_{eval}) - 1 \tag{A.3}$$

The last part of (A.3), $+P(n_{eval}) - 1$ is included in case the evaluation graph is a HCDFG node.

Switch constructs

Switch constructs are represented as CDFGs, G_{CDFG}, and have almost the same flow as the if construct discussed above. One node is a control node of type *switch*. Before arriving to the control node, a condition evaluation node $n_{eval} \in \{G_{HCDFG}|G_{DFG}\}$ is traversed. Depending on the output, the switch node leads the algorithm flow to the selected case node: $n_{case_i} \in \{G_{HCDFG}|G_{DFG}\}$. An example is shown in Fig A.4(b). According to the cyclomatic complexity measure, the number of independent paths is here:

$$P(n_{switch}) = P(n_{eval}) - 1 + \sum_{i=1}^{N} P(n_{case_i}) \tag{A.4}$$



(a) If (b) Switch (c) While

(d) For (e) Parallel (f) Serial

Fig. A.4: Overview of the different CDFGs and combined HCDFGs, on which the cyclomatic complexity values are measured. Between the (HC)DFGs there is a set of data exchange nodes which are here left out for simplicity. The symbols are similar to those presented in Fig A.3.

where N represents the number of cases, i the index to the corresponding node on which the paths are measured.

The same argument goes for the $P(n_{eval}) - 1$ part of (A.4); it is included in case the evaluation graph is a HCDFG node, but else it is omitted.

For loop constructs

For constructs are the most complex of the control structures. Strictly speaking, a for-loop consists of three different parts; the evaluation body, the evolution body and the for-body, n_{eval}, n_{evol}, and $n_{for-body}$ respectively. The control node n_{for}, determines, based on the output from the evaluation graph, whether the flow should go into the for-loop or leave it. The evolution node updates the indexes. Since each iteration of the graph needs to pass through the evaluation and evolution nodes, the number of independent paths is calculated as:

$$P(n_{for}) = P(n_{for-body}) + P(n_{eval}) - 1 + P(n_{evol}) - 1 \tag{A.5}$$

In many cases the evaluation and evolution part of the for-loop are quite simple indexing functions, meaning that $n_{eval} \in \{G_{DFG}\}$, $n_{evol} \in \{G_{DFG}\}$, will leave $P(n_{for}) = P(n_{for-body})$. The For-loop is illustrated in Fig A.4(d).

While and Do-While loops

While and do-while loops are described jointly since it is only the entry to the loop structure that separates them and their cyclomatic complexity are equivalent. The while loops consist of two main parts, the while body $n_{while-body} \in \{G_{HCDFG}|G_{DFG}\}$, and the while evaluation $n_{eval} \in \{G_{HCDFG}|G_{DFG}\}$. This is illustrated in Fig A.4(c). Deciding whether to continue looping is decided by the control node $n_{while} \in \{while\}$ based on the output of the n_{eval}. Similarly to the for-loop, each iteration of the graph needs to pass through the evaluation nodes, so the number of independent paths can be calculated as:

$$P(n_{while}) = P(n_{while-body}) + P(n_{eval}) - 1 \tag{A.6}$$

In many cases the Evaluation part of the while-loop is a set of simple test functions, meaning that $n_{eval} \in \{G_{DFG}\}$, which leaves the $P(n_{while}) = P(n_{while-body})$.

Functions

The goal is to identify the number of independent paths in the algorithm/system. For this, reuse in terms of functions/blocks of code is important. When all independent paths through a function are known, reuse of this function does not change the number of independent paths in the system. From an implementation point of view such functions represent an entity where the paths only need to be implemented once. In HCDFGs, a function/block can be seen as an encapsulated G_{HCDFG}. Therefore the number of independent paths in function/blocks of reused code should only count once. The paths can be calculated as:

$$P(n_{HCDFG_{function}}) = \begin{cases} 0 & \text{if reuse} \\ P(n_{HCDFG}) & \text{else} \end{cases} \tag{A.7}$$

HCDFGs in parallel and serial

Knowing how to handle all the HCDFGs that are identified for reuse (function), together with all the CDFGs, does not give it all. How the hierarchy of graphs should be combined is also of interest. For a parallel combination of two or more HCDFGs/CDFGs, as shown in Fig A.4(e), the increase in the number of independent paths is then additive. The number of paths can be calculated as:

$$P(n_{HCDFG_{Parallel}}) = \sum_{i=1}^{N} P(n_{HCDFG_i}) \tag{A.8}$$

where N represents the number of nodes in parallel, i the index to the corresponding node where the paths are measured.

For serial combination of two or more HCDFGs and/or CDFGs, the number of independent paths is a combination of the independent paths of the involved HCDFGs/CDFGs. Remembering that there always needs to be one path through the system, the number of independent paths in a serial combination, is given as:

$$P(n_{HCDFG_{Serial}}) = \sum_{i=1}^{N} P(n_{HCDFG_i}) - (N-1) \tag{A.9}$$

where N represents the number of nodes in serial, i the index to the corresponding node where the paths are measured.

An example of serial combination is shown in Fig A.4(f). The number of independent paths for the entire algorithm, $(P(n_{HCDFG_{Alg}}))$, is equivalent to the top HCDFG node which includes all the independent paths of its subgraphs.

3.4 Experience Impact

The experience of the designer has an impact on the challenge that he/she is facing when developing a system. A radical example is when a beginner and a developer with ten years of experience are asked to solve the same task. They will not see equal difficulty in the same task, and thereby do not need to put the same effort into the development.

Experience is influenced by many parameters but in this work we only focus on the time the developer has worked with the implementation language and the target architecture.

The impact of experience is a factor that slowly decreases over time: consider a new developer, the experience that he/she obtains in the first months working with the language and architecture improves his/her skills significantly. On the other hand a developer who has worked with the language and architecture for e.g. five years will not improve her/his skills at the same rate by working an extra year. The impact from the experience is therefore not linear but tends to have a negative acceleration or inverse logarithmic nature, with dramatic change in impact in the beginning, progressing towards little or no change as time increases.

In the literature, e.g. [24], many studies try to fit historical data to models. An example of a model is a power function with negative slope or a negative exponential function. From the vast variety of models that has been proposed over the years, the only conclusion that can be drawn is that there are multiple curvatures, but they all appear to have a negative accelerating slope, which tends to be exponential/logarithmic.

In order to get the best possible outset for predicting the implementation effort, it is of vital importance to obtain some data of the developers' experiences, and also how they performed in the past. The parameters involved in the experience curve can then be trimmed to create the best possible fit. However, it has not been the purpose of this work to select

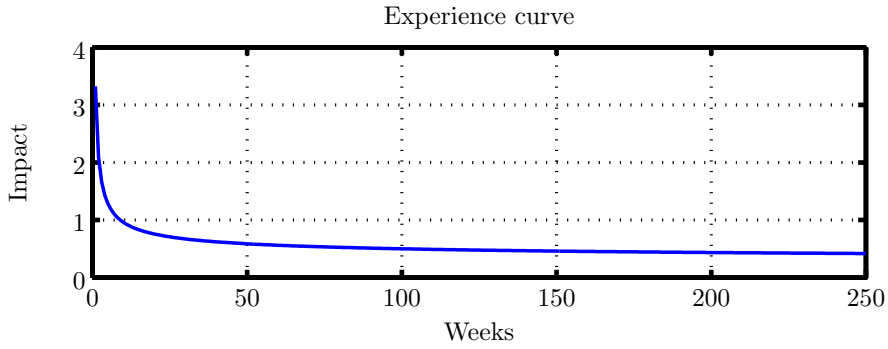


Fig. A.5: An example of how the lack of experience impacts the difficulty the engineers are facing.

the perfect nature for a learning curve nor to evaluate the accuracy of such one. The learning curve will be adapted to the individual developers, and as the model is used in subsequent projects, its accuracy will progressively improve. As a consequence the experience here is only intended as an element in modelling the complexity and thereby a means for more accurate estimates.

For the experiments in this study we have chosen to use the following model:

$$\eta_{experience}(Dev) = \frac{1}{\alpha \log(Experience(Dev) + \beta)} \tag{A.10}$$

where α and β are trim parameters which can be used to optimise the curve to fit reality, $Experience$ is the number of weeks which the developer, Dev, has worked with the language and architecture. Fig A.5 depicts the shape of the experience model.

In this work our initial experiments have shown that setting $\alpha = 1$ and $\beta = 1$ makes our model sufficiently general, and therefore we have not further investigated the tuning of these two parameters.

4 Results

In order to verify the hypothesis, a classical test has been conducted. The test is dual phased and consists of: i) a training phase using a first set

of real-life data, during which the hypothesis is said to be true, and ii) a validation phase during which a second set of real-life data is used to evaluate whether the hypothesis holds true or not.

4.1 Phase One – Training

The real-life data used as training data originate from two different application types that are both developed as academic projects in universities in France. The first application is composed of five different video processing algorithms for an intelligent camera, which is able to track moving objects in a video sequence. The second application is a cryptographic system, able to encrypt data with different cryptographic/hashing algorithms, i.e. MD5, AES and SHA-1. The system consists of one combined engine [25] as well as individual implementations. These projects were selected since they all follow the methodology of using a behavioural specification in C, as a starting point for the VHDL implementation. Common to this data is that none of the developers have made the behavioural specification in C. For the cryptographic algorithms the behavioural specification comes from the standards, and the video algorithms were based on a previous project.

Using the behavioural description as the starting point of the experiment, the exercise consists of studying the relationship between the complexity of the algorithms (as defined in section 3) and the implementation effort (i.e., time) required to implement them in VHDL (including testbed and heuristic tests).

The developers involved in these projects have all been Master and Ph.D. students with electrical engineering backgrounds but no VHDL background other than what they obtained during their studies, see Table A.2. All developers were taught VHDL by other instructors than the authors, but at our university. Table A.3 summaries the training data.

Fig A.6 shows the relation between the implementation effort and the measured complexity for the individual algorithms. Please note that in this graph, the complexity values are not yet corrected for the designers' experience.

A first examination of the data points indicates a possible relation between some of them. However many other points are located far away from any relation. These data are not corrected for the designers' experi-

Table A.1: Line of code, area, and time constraints for the validation data.

Algorithm	SS1	SS2	SS3	SS4	SS5	SS6	Ethernet	App 4
Dev. Time [weeks]	3.6	6.4	2.4	16.4	12	17.2	16	2
LOC-VHDL	994	1195	776	1695	760	2088	3973	232
Slices	564	2212	382	888	372	2171	3372	750
FlipFlops	913	2921	1290	1366	1208	2077	6149	942
LUTs	997	3157	6453	1569	6443	3458	18255	567
Time Constraint. [ns]	112	128	360	112	360	248	696	56

Table A.2: Facts about the developers. Developers for training data (Top) and validation data (Bottom)

Developer	Education	Years in the domain
Dev 1	Ph.D. stud.	0
Dev 2	Stud. (EE)	0
Dev 3	Stud. (EE)	0
Dev 4	Stud. (EE)	0
Dev 5	BSc.EE.	9
Dev 6	MSc.EE.	15
Dev 7	MSc.EE.	9
Dev 8	MSc.EE.	8
Dev 9	MSc.EE.	8

Table A.3: Training Data (Top) and Validation Data (Bottom). Algorithms are related to the developers and their experience at the given time. Complexity is not corrected.

Algorithm	Complexity	Developer	Dev. Exp.
T1	10	Dev 1	2
T2	24	Dev 1	10
T3	12	Dev 1	18
T4	14	Dev 2	1
T5	4	Dev 1	20
MD5	10	Dev 3	1
MD5	10	Dev 4	1
AES	10	Dev 4	8
SHA-1	27	Dev 4	14
Combined	59	Dev 4	14
SS1	25	Dev 6,7	150
SS2	35	Dev 5	150
SS3	17	Dev 5,6,7,8	150
SS4	50	Dev 6	6
SS5	29	Dev 7	3
SS6	25	Dev 5,6,7	3
Ethernet app	60	Dev 5,6,7,8,9	150
App 4	9	Dev 6	150

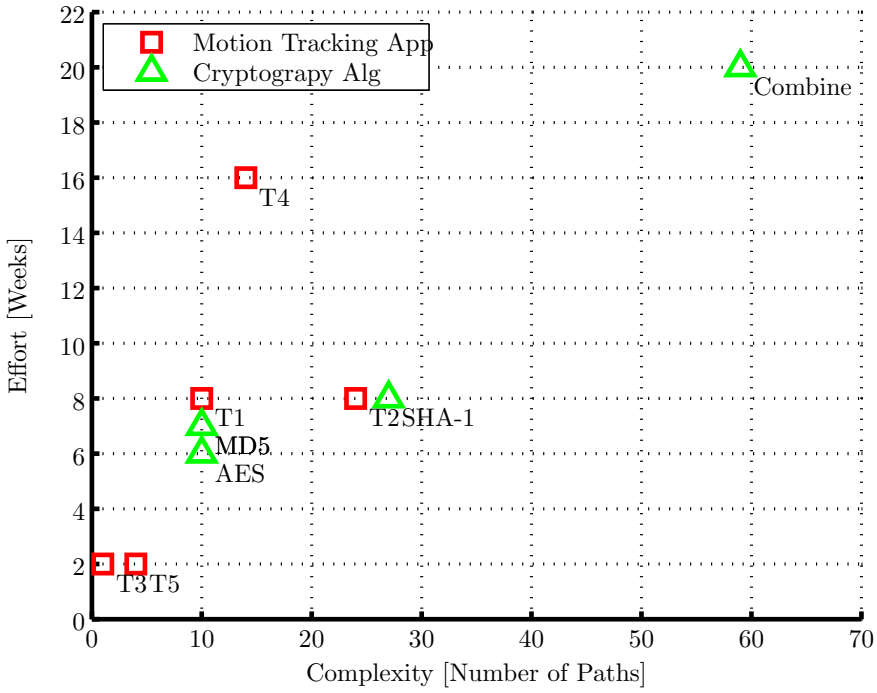


Fig. A.6: Relation between the implementation effort [number of weeks] and the not corrected complexity (as defined in section 3).

ence and, as earlier mentioned, we strongly believe that the experience of the individual designer has a non-negligible influence on the development time. If we inspect the data more thoroughly, it is clear that the points of greatest divergence, are those implementations where the developers have very limited knowledge and experience with the VHDL language.

Applying the proposed (A.10) (non-linear) experience transform onto the data, results in a significantly different picture, as depicted in Fig A.7. A clear trend toward a relation is now visible in the plotted data. From the COCOMO II project [8] it is known that the relationship between the implementation time and the complexity measure (in their case lines of code, LOC) can be expressed as a power function with a weak slope. We showed its nature in (A.1), and with correction for experience it becomes:

$$Effort = A \cdot (\eta_{experience}(Dev) \cdot P(n_{HCDFG_{Alg}}))^b \qquad (A.11)$$

The parameters A and b are found, via a least square (LS) fit on our training data, to be $A = 0.226$ and $b = 1.103$. In Fig. A.7 the dashed line illustrates the relationship, with the parameters given above.

4.2 Phase Two – Validation

After having elaborated on a model based on the training data, we proceeded with the validation of its correctness. For this, a new set of data provided by ETI A/S, a Danish SME, is used. The dataset originates from a networking system and consists of Ethernet applications that have been implemented on an FPGA, as well as corresponding testbeds. This Ethernet application is part of an existing system with which it requires interaction. Table A.1 shows additional implementation information with regards to these applications. The system is a real-time system with hard time-constraints and all algorithms were implemented as to meet these constraints. Similar to the training data, the development flow for this application has been as follows: a behavioural C++ model of the application has been constructed before the implementation on the FPGA architecture. The behavioural model has been developed by developers separate to those undertaking the implementation. The developers responsible for the implementation have obtained their skills in VHDL from a professional course with no relation to our university in Denmark.

The time spent on the implementation process covers: the design and implementation of the VHDL code of the functionalities and testbed as

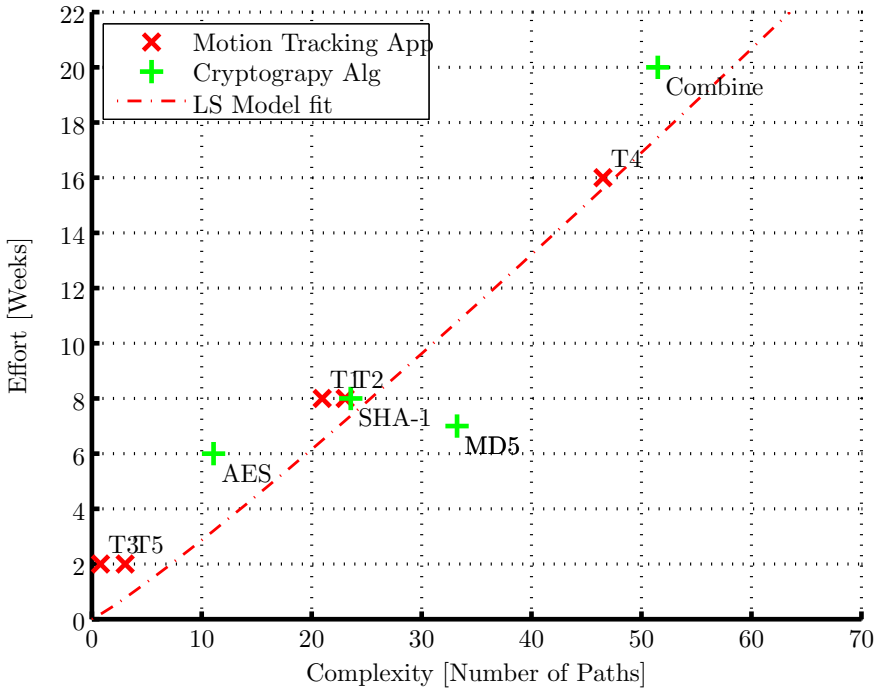


Fig. A.7: Relation between the implementation effort [number of weeks] and the complexity corrected according to the designers' experience model as shown in Fig A.5.

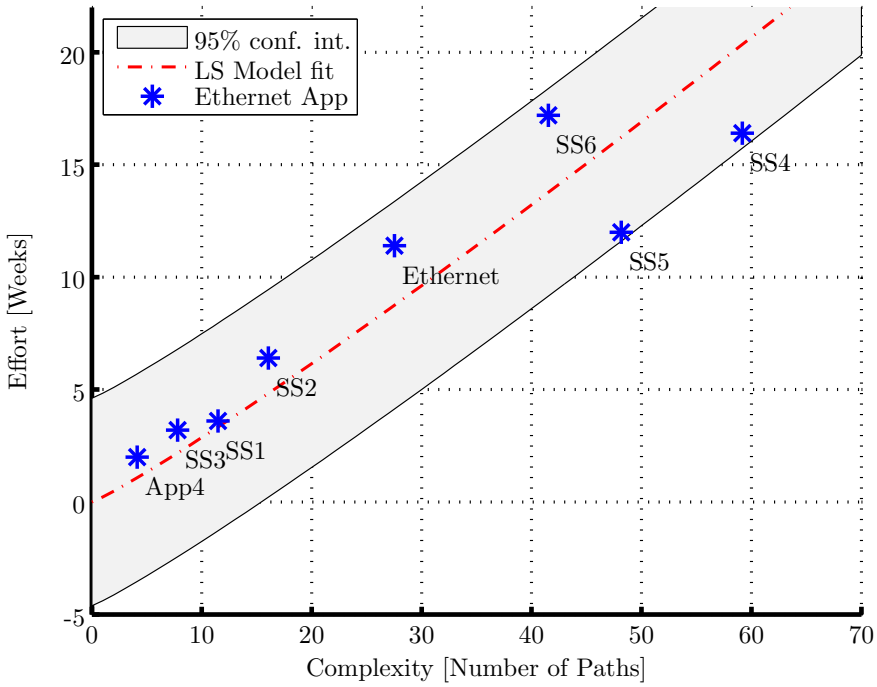


Fig. A.8: Validation data plot: relation between implementation effort [number of weeks] and complexity, corrected according to the designers' experience model.

well as the tests of the different modules in the applications. This data is shown in the lower part of Table A.3. The time data originate from the company's internal registration for the project, and correspond therefore to the effective time used.

The relation between implementation effort and complexity is plotted in Fig A.8. It can be seen that this data, corrected for the designers' experience (*) closely follows the model derived from the training data (dashed line). Fig A.8 also shows the 95% confidence interval, indicating that with 95% confidence, future predictions of implementation effort will lie within this, given that the model holds true.

Comparing the predicted effort (dashed line) to the real effort (*), indicates that there is an estimation error. The values are also shown in Tabel A.4 The average estimation error is 0.2 weeks with a variance of 8.

In the next section we discuss the validity of the model.

Table A.4: Development time and estimated development time measured in weeks together with the error.

Algorithm	Dev. Time	Est. Dev. Time	Error
SS1	3.6	3.3	0.3
SS2	6.4	4.8	1.6
SS3	3.2	2.2	1
SS4	16.4	20.3	-3.9
SS5	12	16.2	-4.2
SS6	17.2	13.8	3.4
Ethernet app	11.4	8.8	2.6
App 4	2	1.1	0.9
Mean (Variance):			0.2 (8)

4.3 Validity Discussion

Estimating the effort required in implementing an algorithm into hardware involves many parameters. We discussed a number of these parameters in section 1.2, but could not include them all in this study. The proposed model is therefore devised from the idea of the relation between implementation effort and number of linear independent paths.

To validate the model, a classical two-phased hypothesis test has been performed and the validity of this test depends on the following important factors: i) the independence between training and validation data ii) the volume and variety of the experiments.

In the first instance, not only different applications were used for training and validation data, but in addition the developers had no relation in terms of education, nationality, work, etc. Moreover, the validation data has not been measured before the model was trained. All this strengthens the validity of the results. The only potential connection is that some of the developers who have been involved in the implementation of the training and validation data have also been included within those interviewed. However, this accounts for a minority and we see this as a minimal risk.

Secondly, we should ideally have had a large volume and variety of experimental data for training and validation. However, our set of data originates from a single company and a few developers. So strictly speaking we can only conclude that this model applies to the specific SME setup involved in the study and partially to the academic environment studied.

In order to generalise our model, more cases of validation are needed. However, obtaining all the statistical data for this new methodology is time consuming. We would therefore like to remind the reader that this paper proposes a methodology for estimating implementation effort and the validation of the model concentrates on illustrating its usefulness. Looking at the graphs, we can determine a clear trend in the results. The curve identified in the training data are sustained for the validation data as well: they both fall in line with the underlying rationale, and we are quite confident in the strength of the proposed model.

The results clearly show the necessity for the proposed correction function; the proposed logarithmic nature works well, even though the correction function has not been trimmed to fit the individual developers due to the lack of available data. In this light, our approach must be seen as the engine of a global methodology for the management of design projects, that impose a systematic registration of man-power. With such a registration, a database of the developers' experience can easily be constructed and the correction function can be trimmed to fit the companies' individual designers. Several iterations of this process would provide convergence towards a more precise estimation of the implementation effort.

The limited data set on which the model is constructed also limits the complexity window to which this model can be applied: having no algorithm with a corrected complexity value larger than 51, extrapolating the model further would weaken the current conclusion. More training data, from larger and more varied projects would allow for a more refined model.

Nevertheless, the results described in this paper are very encouraging with all the real-life cases that we have examined and we are reasonably confident that this model can easily be applied to other types of applications.

5 Conclusion

The contribution presented in this paper is a metric-based approach for estimating the time needed for hardware implementation in relation to the complexity of an algorithm. We have deduced that a relationship exists between the number of linear independent paths in the algorithm and the corresponding implementation effort. We have proposed an original solution for estimating implementation effort that extends the concept of the cyclomatic complexity.

To further improve our solution, we developed a more realistic estimation model that includes a correction function to take into account the designer's experience.

We have implemented this solution in our tool Design Trotter of which the input is a behavioural description in C language and output is the number of independent paths. Based on this output and the proposed model, we are able to predict the required implementation effort. Our experimental results, using industrial Ethernet applications confirmed that the data, corrected for the designers' experience, follows the derived model closely and that all data falls inside its 95% confidence interval. Using this method iteratively paves a way for an implementation effort estimator of which the accuracy improves continuously after each project.

References

[1] D. Blaza, "Embedded systems design state of embedded market survey," Embedded.com – CMP Media, Tech. Rep., 2006, ftp://ftp.embedded.com/pub/ESD%20SubscribSurvey/2006%20ESD%20Market%20Study.pdf.

[2] R. Nass, "An insider's view of the 2008 embedded market study," Embedded.com – CMP Media, Tech. Rep., 2008, http://www.embedded.com/products/softwaretools/210200580.

[3] "Workshop for danish smes developing embedded systems co-organized by the danish technological institute, the center for software defined radio (csdr) and the center for embedded software systems (ciss)," Nyhedsmagasinet Elektronik & Data, Nr.1 2008, Aarhus, Denmark, 2008.

[4] O.-H. Kwon, "Keynote speaker: Perspective of the future semiconductor industry: Challenges and solutions," DAC, San Diego, CA, USA, 2007. [Online]. Available: http://www.dac.com/44th/44talkindex.html

[5] S. McConnell, *Software Estimation – Demystifying the Black Art*. Wasthington: Microsoft Press, 2006.

[6] R. Abildgren, A. Saramentovas, P. Ruzgys, P. Koch, and Y. L. Moullec, "Algorithm-architecture affinity – parallelism changes the picture," in *Proceedings on the Design and Architectures for Signal and Image Processing 2007 Workshop*, 2007.

[7] M. A. Honey, "The interview as text: Hermeneutics considered as a model for analysing the clinically informed research interview," *Human Development*, vol. 30, pp. 69–82, 1987.

[8] B. W. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece, *Software Cost Estimation with COCOMO II*. Prentice Hall, 2000.

[9] B. W. Boehm, *Software Engineering Economics*. Prentice Hall, 1981.

[10] T. Jones, *Programming Productivity*. New York: McGraw–Hill, 1986.

[11] G. Low and D. Jeffery, "Function points in the estimation and evaluation of the software process," *Software Engineering, IEEE Transactions on*, vol. 16, no. 1, pp. 64–71, January 1990.

[12] A. J. Albrecht, "Measuring application development productivity," in *Proc. IBM Applications Development Symp.*, 1979.

[13] D. Jeffery, G. Low, and M. Barnes, "A comparison of function point counting techniques," *Software Engineering, IEEE Transactions on*, vol. 19, no. 5, pp. 529–532, May 1993.

[14] W. Fornaciari, F. Salice, U. Bondi, and E. Magini, "Development cost and size estimation starting from high-level specifications," in *Proceedings of the ninth international symposium on Hardware/software codesign*, 2001, pp. 86–91.

[15] T. J. McCabe, "A complexity measure," *IEEE Transaction on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, December 1976.

[16] D. L. Lanning and T. M. Khoshgoftaar, "Modeling the relationship between source code complexity and maintenance difficulty," *Computer*, vol. 27, no. 9, pp. 35–40, 1994.

[17] T. J. Walsh, "Software reliability study using a complexity measure," in *Proceedings of the National Computer Conference*, New York, 1979.

[18] S. P. VanderWiel, D. Nathanson, and D. J. Lilja, "Complexity and performance in parallel programming languages," in *HIPS '97: Proceedings of the 1997 Workshop on High-Level Programming Models and Supportive Environments (HIPS '97)*. Washington, DC, USA: IEEE Computer Society, 1997, p. 3.

[19] M. Mastretti, M. Busi, R. Sarvello, M. Sturlesi, and S. Tomasello, "Vhdl quality: synthesizability, complexity and efficiency evaluation," in *EURO-DAC '95/EURO-VHDL '95: Proceedings of the conference on European design automation*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1995, pp. 482–487.

[20] I. Feghali and A. H. Watson, "Clarification concerning modularization and mccabe's cyclomatic complexity," *Communication of the ACM*, vol. 37, no. 4, pp. 91–94, 1994.

[21] B. H. Sellers, "Modularization and mccabe's cyclomatic complexity," *Communication of the ACM*, vol. 35, no. 12, pp. 17–19, 1992.

[22] Y. L. Moullec, N. B. Amor, J.-P. Diguet, M. Abid, and J.-L. Philippe, "Multi-granularity metrics for the era of strongly personalized SOCs," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2003.

[23] Y. L. Moullec, J.-P. Diguet, N. B. Amor, T. Gourdeaux, and J.-L. Philippe, "Algorithmic-level specification and characterization of embedded multimedia applications with design trotter," *Journal of VLSI signal Processing*, vol. 42, pp. 185–208, 2006.

[24] A. Heathcote, S. Brown, and D. J. K. Mewhort, "The power law repealed: The case for an exponential law of practice." *Psychonomic Bulletin & Review.*, 2000.

[25] S. Ducloyer, R. Vaslin, G. Gogniat, and E. Wanderley, "Hardware implementation of a multi-mode hash architecture for MD5, SHA-1 and SHA-2," in *Proceedings on the Design and Architectures for Signal and Image Processing 2007 Workshop*, 2007.

Paper B

Real-Time Aware Hardware Implementation Effort Estimation

Rasmus Abildgren, Jean-Philippe Diguet, Guy Gogniat, Peter Koch, and Yannick Le Moullec

This paper is a techical report from
Aalborg University,
2010.

The layout has been revised.

Abstract

This paper presents a structured method and underlying models for estimating the hardware implementation effort of hard real-time constrained embedded systems. We propose an optimization model which takes some of the most common optimization techniques into account as well as the order in which they should be applied. We suggest a set of two metrics used to characterise the effects of optimisations: one expressing how hard it is to reach an implementation satisfying the real-time constraints for the implementation, and another one to reflects how the distribution of parallelism in an algorithm influences the impact of the optimisations. Experimental results do not show an unambiguous result. However, for most algorithms our approach enables the estimation of the hardware implementation effort for hard real-time constrained applications.

1 Introduction

The need for continuous innovation combined with growing complexity, increased product release frequency, increasing time-to-market pressure and fierce competition, make the task of project managers working in the embedded systems industry more and more challenging. For example, the 2009 Embedded Market Study [1] reports that 63% of the projects were not finished on schedule and that the average lateness is 4.4 months.

In this context, accurate development time estimates are an essential tool which can make the difference between success and failure. However, obtaining such estimates is not a trivial task since development time depends on many factors, including both technical (hardware and software e.g. products built upon new platforms, area/time/energy constraints), human (e.g. skills, mood of the developers), and managerial aspects.

Whereas methods and techniques are readily available for estimating the development time of software executing on GPPs [2], mainly for desktop applications, to the best of our knowledge little efforts have been carried out in the hardware domain. Working with a systematic approach for estimating the development time for different projects requires a certain maturity in the organisation. Many small and medium sized enterprises (SMEs) usually do not have such priories, although they would benefit from it. Many SMEs perform ”ad hoc” estimations (e.g. based on expe-

rience or intuition), but in many cases these ad hoc approaches do not provide accurate estimates, which in turn means delayed projects. This work proposes a method and tools which offer a systematic and structured approach for estimating more precisely the implementation effort.

1.1 Our Prior Work

The work presented in this paper is part of a larger effort aiming at improving this situation [3]. In this paper we describe our contribution regarding the problem of estimating the hardware implementation effort (in terms of development time) for real-time constrained applications. This contribution is an extension of what is summarized below.

In [3] it has been shown that every path in the algorithm(s) that the designer must implement adds to the development time and that the complexity of a design can be expressed by the number of independent paths in the algorithm(s). It has also shown that, when the experience of the designer is taken into account, a relation between the number of independent paths and development time exists and that it is possible to estimate the hardware implementation effort (in terms of development time) of applications.

However, in many cases the implementation can become very challenging when (hard) real-time constraints need to be fulfilled. Typically in such cases only a limited number of "implementational tracks" lead to a (or sometimes the) satisfying solution. This can be illustrated as in Fig. B.1, where only one implementation track (the thick red line) satisfying the contraints. The idea on which this work builds upon is that real-time constraints make the implementation more difficult and that, in order to fulfill these constraints, designers need to perform certain optimizations in a certain order. Identifying a or the suitable track(s) adds to the overall development time since extra efforts must be spent at evaluating and applying the right combination (i.e. type and order) of optimization techniques. In order to take these considerations into account, we propose to complement and augment our prior work with several extensions. These extensions are described in Section 1.2.

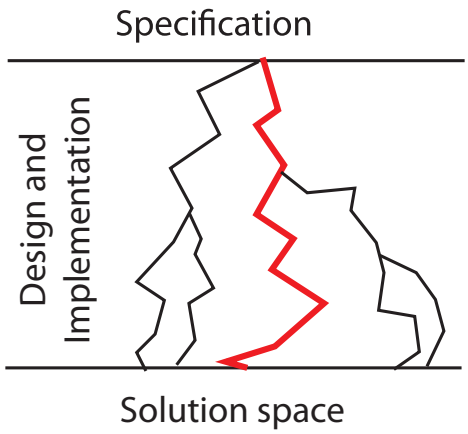


Fig. B.1: The task of designing and implementing an algorithm can be seen as going from a specification to a solution. Typical more than one solution will satisfy the same specification. However, when constraints are introduced, the solution space narrows down and only a few or one "implementation tracks" (depicted as the thick red line) will result in a satisfying solution.

1.2 Contributions

One major contributor to the overall development time is the implementation effort. The contributions presented in this paper are i) a method and ii) a set of underlying models aiming at estimating the implementation effort, measured in time, of real-time constrained embedded applications. We propose an optimization model which takes some of the most common optimization techniques into account as well as the order in which they should be applied. An essential contribution of this work is a set of two metrics used to characterise the effects of optimisations. The first one expressing how hard it is to reach an implementation satisfying the real-time constraints for the implementation. The second one reflects how the distribution of parallelism in an algorithm influences the impact of the optimisations.

The remainder of the paper is organized as follows: section 2 introduces related works. Section 3 details the proposed methodology and section 4 details the metric for the distribution of the parallelism. Subsequently, section 5 presents and discusses the experimental results ob-

tained. Finally, section 6 concludes the paper.

2 State of the art - Effort Estimation

To the best of our knowledge, very few works address the problem of estimating the hardware implementation effort of hard real-time constrained applications. On the other hand, there exist several approaches for estimating the software implementation effort, some of them providing ideas and directions for the hardware oriented ones. Thus, in this section we start by reviewing the most relevant approaches for estimating the software implementation effort and proceed with the few existing approaches for hardware implementation effort estimation.

Some of the most known and used tools for estimating the software implementation effort are the COCOMO project [2], function point [4], and SPQR/20 [5]. They all build upon the same concept: firstly, in order to quantify certain properties of an algorithm, a measure or set of measures is defined. Secondly, a model describing the relation between the measure(s) and the implementation effort is derived.

The core idea in COCOMO (COnstructive COst MOdel) [2] is that the effort mainly depends on the project size, i.e., $Effort = A \cdot size^b$ where A and b are adjustable parameters which must be trained in order to reflects factors such as manpower, experience of the developers, etc. The remaining parameter in the equation, the size of a project, can be measured by means of e.g. Lines of Codes (LOC); however, this is subject to criticism and thus other measures have been proposed like function point.

Function point [4] consists of two main stages: the first stage consists in counting and classifying the function types of the software: identified functions are weighted to reflect their complexity, which in practice is left to the developers's perception. The second stage is the adjustment of the function points based on 14 parameters which are tuned according to the characteristics of the application and of its environment. Subsequently, the function points are converted into a LOC measure based on an implementation language-dependent factor, which in turn can be used as an implementation effort estimation metric.

SPQR/20 (Software Productivity, Quality and Reliability with regard to 20 influencing factors) has been proposed as a less heuristic-oriented

variant of function point; experimental results [6] suggest that it can provide the same accuracy than function point while being simpler to work with.

Publications dealing with the estimation of hardware implementation effort are far less abundant than those dealing with software. Considering the context of the present work, interesting approaches include VHDL function point [7] and cost models such as [8]. Several other publications such as [9] compare actual hardware implementation efforts for different design methodologies but do not provide any systematic method to estimate those efforts.

VHDL function point, presented in [7], builds upon the idea of function points analysis and is modified to work with VHDL code. The approach consists in counting the number of internal I/O signals and components, and classifying these counts into levels. From there, a function point value related to VHDL is extracted. Experimental results considering the number of source lines in the LEON-1 processor project yields predictions which are within 20% of the real size. However, estimating the size does not always give an accurate indication of the implementation difficulty, especially when the application is subject to real-time constraints.

[8] introduces a cost model with the objective of understanding current Product Development Cycles (PDC) and evaluating the impact of new technologies on these PDC. In particular, the authors focus on cost and product development time and propose a PDC known as One Pass to Production (OPP) which takes both software and hardware aspects of a complete system into consideration. Although promising, their approach is very specific (they consider a FPGA-based NOC backbone) and the numerous assumptions made by the authors (e.g. regarding the number of required engineers) make it challenging to see how their approach could be made sufficiently generic to be applied to much more varied types of applications.

We can safely conclude that there is currently a lack of suitable and systematic methods and tools for estimating the hardware implementation effort for real-time constrained applications. In what follows we present our contribution to improve this situation.

3 Methodology

In [3] it has been shown that the hardware implementation effort can then be modelled as

$$Effort = A(\eta(Dev) \cdot P(alg))^b \tag{B.1}$$

where η reflects the experience of the developer Dev, $P(alg)$ is the number of independent path in the algorithm alg, and A and b are trim parameters.

Experimental results have shown that it is possible to estimate the hardware implementation effort, expressed as the development time, of applications and that the proposed model is able to estimate the need implementation effort with a confidence interval of 95%. However, this approach is not specifically targeting real-time constrained applications and is therefore not suitable for this type of application.

Since we in this work want to take hard real-time constraint into account, we propose a method which adds a parameter $\tau(t_c)$ expressing the difficulty or hardness of reaching an implementaion which meets the time constraint, t_c. This parameter we will call implementation hardness and therefore the effort can be modeled as:

$$Effort = A(\eta(Dev) \cdot P(alg) \cdot \tau(t_c))^b \tag{B.2}$$

The underlying idea is that as far the execution time t_{exec} is from t_c the more difficult it will be to fullfil t_c. Whenever t_c is not met, optimizations have to be performed. However, modeling optimizations and their impact is not a trivial task for a designer; therefore, in what follows, we propose a method and a set of models which reflect the most common cases.

3.1 Real-Time Constraint

When optimizing the implementation to meet a real time constraint, the optimization strategies can be many fold. For a developer, the optimization strategy is very much application dependent but also depends on his experience and on his analytical thinking. Optimizations can fall into two different domains; spatial and temporal. Optimizations in the spatial domain include algorithmic parallelism exploitation on multiple functional

units. For the temporal domain, different optimization techniques exist such as chaining and pipelining.

Typically, the type of optimization to be performed in the temporal domain is chosen depending on a) data/control dependencies in the algorithm and b) the constraint type:

- Throughput (pipelining)
- Latency (chaining)

Both type of constraints can benefit from parallelism exploration. Usually when analyzing an algorithm, for e.g. parallelism, the measure applied will indicate the potential of exploiting the entire parallelism in the algorithm, as for example with the measure γ [10]. Performing a straight manual implementation of an algorithm will usually not result in a complete exploitation of the parallelism. Either because it is not necessary or because the designer has omitted optimizations which could have a significant impact on the exploitation of the algorithm's inherent parallelism.

An illustration of the overall optimisation approach is shown in Fig. B.2. The starting point will usually be a sequential version of the algorithm. The developer chooses in which order he/she performs one or several of the different optimisations strategies. The order and types of strategies will vary from algorithm to algorithm. To generalise our approach we constrain the optimisation strategy to follow the template denoted by the thick line in Fig. B.2. This limits the overall strategy to complete the parallelism optimisation strategy before starting the chaining strategy. We will not consider pipelining optimisations further in this paper.

In order to guide the designer in the exploration of the parallelism, this work considers a fully spatially parallelized algorithm as an extreme. Similarly, a complete chained implementation is also considered as extremes, these extremes indicate the bounds of how much speedup can be obtained when applying the respective type of optimization, without rewriting the algorithm.

Furthermore, not knowing the exact strategy that a development engineer is following, but knowing which options he/she has, our hypothesis is that it is possible to estimate the minimum number of optimizations required in order to fulfill a real-time constraint. This, in turn, provides

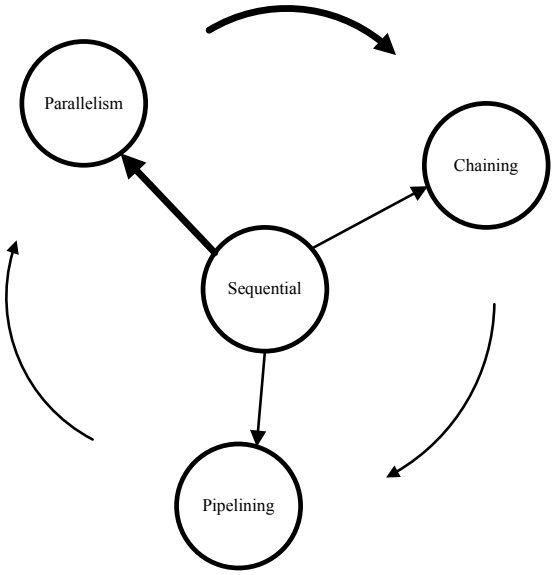


Fig. B.2: The overall optimisation approach where the starting point usually will be the sequential version of an algorithm. The developer freely chooses in which order he/she performs the different optimisations strategies. For our approach we constrain the optimisation strategy to follow the thick line.

useful information which can be converted into the implementation hardness parameter, τ, of Eq. B.2 for estimating the required implementation effort.

It is therefore important to know how many optimizations inside the different categories should be applied in order to fulfill the time constraint, t_c. The next section describes the concept of estimating the execution time on basis of the number of optimisations.

3.2 Optimisation Dependent Execution Time Estimation

Every optimisation yields a certain speed-up to the execution time. Fig B.3 shows the execution time of an algorithm with different numbers of optimisations for the different optimisation categories. This is illustrated by the relation between the number of applied optimisations (represented by the small vertical lines) and the resulting execution time (t_{exec}) for several optimisation strategies (parallelization and parallelization+chaining).

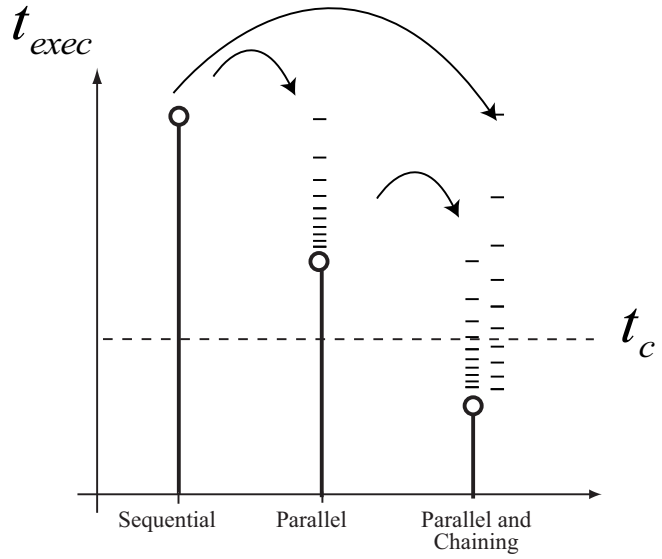


Fig. B.3: Relation between the number of applied optimisations (represented by the small vertical lines) and the resulting execution time, t_{exec}, for several optimisation strategies (parallelization and parallelization+chaining). The reduction of the execution time gets smaller as the number of optimisations increases for a certain strategy (represented by the spacing between the small vertical lines). In order to arrive at an execution time equal to or smaller than the time constraint, t_c, several possible paths exist which depend on a combination of different optimisation strategies and the number of applied optimisations for each strategy.

The reduction (per optimisation) of the execution time gets smaller as the number of optimisations increases for a certain strategy. This is represented by the spacing between the small vertical lines. In order to arrive at an execution time equal to or smaller than the time constraint (t_c) represented by the dashed line, the designer can choose between several possible paths. An optimization path is the number of optimizations performed in the parallelization category followed by the number of optimizations performed by chaining. The number of optimisations in the different categories can vary since there can be more than one path satisfying the time constraint.

Therefore, it is necessary to know an estimate of the execution time for different optimization paths. In the following we describe how to estimate the execution time for the non-optimized case and the three

different optimisation cases:

Case 0: No optimization (sequential execution)

The most simple case is the sequential execution. To calculate the estimate, t_{exec}, we use the following equation:

$$t_{exec}(0) = NOP_{atomic} \frac{1}{\hat{f_{arch}}} \tag{B.3}$$

where NOP_{atomic} denotes the number of operations in the sequential algorithm and $\frac{1}{\hat{f_{arch}}}$ the time for executing one operation. In this work we assume that all operations can be considered as atomic and therefore have the same execution time.

Case 1: Parallel optimization

The estimated execution time, $t_{exec}(NOO_{PAR})$, when applying a certain number of parallelization optimisations, NOO_{PAR}, can be expressed as

$$t_{exec}(NOO_{PAR}) = \frac{NOP_{atomic}}{\gamma_{impl}(NOO_{PAR})} \frac{1}{\hat{f_{arch}}} \tag{B.4}$$

where $\gamma_{impl}(NOO_{PAR})$ expresses the degree of speed-up obtained with NOO_{PAR} number of optimisation. This can be calculated as:

$$\gamma_{impl}(NOO_{PAR}) = \frac{NOP_{atomic}}{NOP_{atomic} - NOP_{optimised}(NOO_{PAR})} \tag{B.5}$$

where $NOP_{optimised}(NOO_{PAR})$ expresses the reduction in executed operations in the critical path when NOO_{PAR}, number of optimizations, are applied. How to obtain this estimate is further discussed in section 3.3.

All in all this gives:

$$t_{exec}(NOO_{PAR}) = (NOP_{atomic} - NOP_{optimised}(NOO_{PAR})) \frac{1}{\hat{f_{arch}}} \tag{B.6}$$

Case 2: Chaining

Similarly, for chaining we can express the estimated execution time, $t_{exec}(NOO_{Chain})$, as:

$$t_{exec}(NOO_{Chain}) = (NOP_{atomic} - NOP_{optimised}(NOO_{Chain})) \frac{1}{\hat{f_{arch}}} \tag{B.7}$$

where NOO_{Chain} denotes the number of applied chaining optimizations. Please note that, $\hat{f_{arch}}$, the frequency of the architecture will typically change when creating larger operators.

Case 3: Combined

Combining the parallelized and chained cases will leave us with the following equations:

$$t_{exec}(NOO_{PAR}, NOO_{Chain}) = \left(\frac{NOP_{atomic}}{\gamma_{impl}(NOO_{PAR})} - \frac{NOP_{optimised}(NOO_{Chain}|NOO_{PAR})}{\varphi(NOO_{PAR})^{-1}} \right) \frac{1}{\hat{f_{arch}}} \tag{B.8}$$

where $\varphi(NOO_{PAR})$ is a parallelism distribution measure which takes the fact that a chaining optimisation in the parallel context does not necessarily result in a reduction of the execution time. We discuss this later in section 4.

3.3 Optimisation Impact Estimation

When implementing an algorithm containing loops, different loops have different numbers of iterations. Typically, loops with larger numbers of iterations contribute more to the execution time of the algorithm than loops with small numbers of iterations. Optimizing an operation in a loop with a large number of iterations yields a larger reduction of the execution time. Assuming that the effort required to perform an optimisation does not change with the number of iterations, processing the loops with the largest number of iterations first, pays a larger impact

on the reduction of the execution time for a given implementation effort. It is therefore essential to take this into account when estimating the optimisation impact on the execution time.

The real impact of an optimisation on algorithms that include loops can not be known without deep inspection of the algorithm; however, an approximation would be beneficial. We therefore propose a measure approximating that. The requirements for defining such a measure include that it should reflect the number of executed operations compared to the number of operations that need to be implemented. In Fig. B.4 a random algorithm containing loops is considered. The figure shows the relation between the number of optimisations and the corresponding reductions in the executed number of operations. The optimisations are ordered according to their impact on the execution time of the algorithm. The solid line represents the real impact. The dashed line, the average and the dotted line, a first order logarithmic based approximation. The real line corresponds to the case where the optimisations are fully prioritised according to their impact, the average line corresponds to the mean impact of a random optimisation strategy.

One interesting point in the graph in Fig B.4 is the end point. The number of operations which can be parallelized as well as the number of operations which can be chained limit the possible number of optimisations. We denote the maximum number of optimisations for the parallel case as:

$$|NOO_{PAR}|_{max} = NOP_{impl} - CP_{impl} \tag{B.9}$$

where NOP_{impl} represents the number of implemented operations and CP_{impl} the number of implemented operations which are present in the critical path. These numbers are different from NOP and CP when loops are present since the operations inside a loop are executed several times. Similarly to the measure in Eq. B.9 a measure, $NOP_{optimised}(|NOO_{PAR}|_{max})$, for the maximum number of executed optimised operations can be calculated. The ratio between these two measures reflects the average impact of the loops present in the algorithm when taking the parallelism into account. This will be the slope of the average line in Fig. B.4.

A similar approach is used for the chaining case except that the maximum number of optimisations is calculated as:

$$|NOO_{Chain}|_{max}(NOO_{PAR}) = NOP_{impl} - P(NOO_{PAR}) \quad \text{(B.10)}$$

where $P(NOO_{PAR})$ denotes the number of paths in the algorithm, which is further detailed in section 4.

It turns out to be difficult to obtain a good and stable first order approximation of the impact of the loops in the algorithm based on the limited number of data which we have available. We have therefore decided to use the average as a measure for the impact of an optimisation which can be calculated as:

$$NOP_{optimised}(NOO_{PAR}) = \frac{NOP_{optimised}(|NOO_{PAR}|_{max})}{|NOO_{PAR}|_{max}} NOO_{PAR} \quad \text{(B.11)}$$

and

$$NOP_{optimised}(NOO_{Chain}|NOO_{PAR}) = \frac{NOP_{optimised}\left(|NOO_{Chain}|_{max}(NOO_{PAR})\right)}{|NOO_{Chain}|_{max}(NOO_{PAR})} NOO_{Chain} \quad \text{(B.12)}$$

4 Metrics

4.1 Metric of distribution of parallelism

When chaining operators, the impact on the execution time depends on whether the optimisations are done in the critical path or in other paths. For this work we expect that the developer has carefully analysed the algorithm and is only optimising where it is most feasible, i.e. in the critical path.

However, chaining operations in the critical path can lead to a situation where the path which was originally the critical one is reduced, due to the optimisations, so another path becomes the the longest one. Fig B.5 shows three different examples, all having 15 nodes and a critical path of 5, which gives a speedup measure, $\gamma = 3$. Fig B.5(a) shows an example where the initial critical path (grey) has the same length as

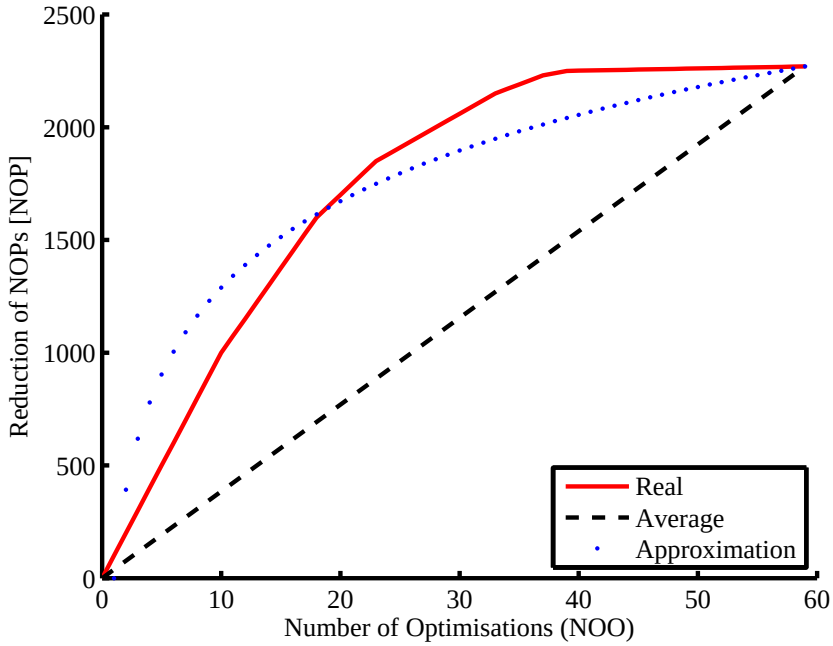
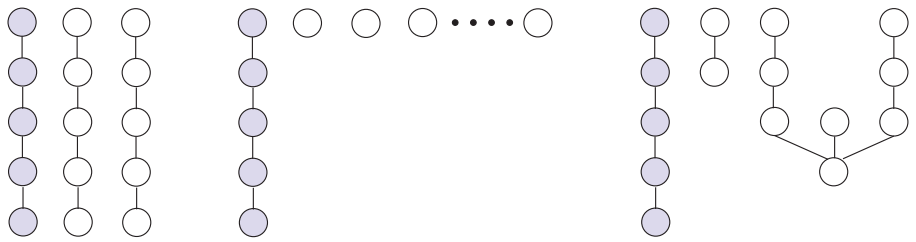


Fig. B.4: The figure shows the impact (in terms of reduction) of optimisations in an algorithm with operations in different loops and also outside loops. The operations are optimised following the order of their impact on the execution time of the algorithm. The solid line represents the real impact. The dashed line, the average and the dotted line, the approximated. The real line corresponds to the case where the optimisations are fully prioritised according to their impact, the average line corresponds to the mean impact of a random optimisation strategy.



(a) Extreme 1 (b) Extreme 2 (c) Average

Fig. B.5: Illustration of the distribution of the parallelism in the algorithm. Fig B.5(a) and B.5(b) shows the two extremes, with either the critical paths (grey) being comparable to the other paths (denoted highly distributed case) or completely unique (denoted narrow case). Fig B.5(c) shows a more average case. All examples have 15 nodes and a critical path of 5, which will give a speedup measure, $\gamma = 3$. When operations get chained, the different cases lead to different reductions of the critical path. This makes it difficult to predict the reduction of the critical path per chaining optimisation. This calls for a metric indicating the distribution of the parallelism.

the two other paths in the algorithm (highly distributed case). Chaining two operations in the critical path will change the longest path to one of the two others. Opposite to this, Fig B.5(b) shows an example where the initial critical path is significantly longer (narrow case), which means that chaining operations in this case will lead to a reduction of operations in the initial critical path. In between, Fig B.5(c) shows a more average example.

Knowing the graph would make it possible to derive the exact reduction of the algorithm's critical path with a specific number of chaining optimisations. However, not knowing the graph but only the average speedup, γ, the number of operations and the length of the initial critical path makes it challenging to predict this reduction.

In order to obtain a more sufficient estimate of the effect of an average chaining optimisation, we propose a metric which considers the distribution of the parallelism in the algorithm.

It is desirable that such a metric has the following properties: in case of a highly distributed parallelism (see Fig B.5(b)), i.e. many paths in the algorithm, the value of the metric should converge towards one. In case the distribution is "narrow" (see Fig B.5(a)), i.e. the number of paths is equivalent to the speedup, the metric should give a value close to zero.

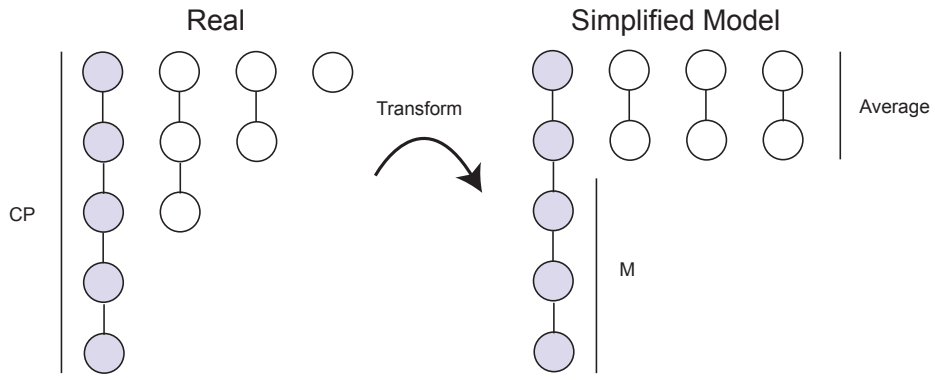


Fig. B.6: Every graph can be transformed into a graph which can be treated as the two extremes of Fig B.5. M denotes the length of the critical path which is longer than the average of the off critical paths length.

Most graphs will not fall into the two extremes from Fig B.5, but will be more like the average case. In order to obtain the metric of the the average contribution of chaining optimisations, we propose a mechanism with which any graph can be transformed and handled as a combination of the two extremes. The transformation is illustrated in Fig B.6. The mechanism is as follows: keep the critical path fixed and substitute the off critical paths (i.e. all paths excluding the critical one) with paths having their average length.

Doing this transformation brings us to a simplified problem where we first can handle chaining as the narrow distributed case, and second as the highly distributed case. It is given that the impact of the chaining optimisation will always be better or equal to this simplified model, with this number of paths. An optimisation following this model is shown in Fig B.7.

In order to handle that graph as the narrow distributed case, it is important to know how large the difference between the critical path and the off critical paths is. To do so we utilize $P(NOO_{PAR})$, the number of paths of the parallel-optimised algorithm. The difference between the critical path and the average of the off critical paths can then be calculated as:

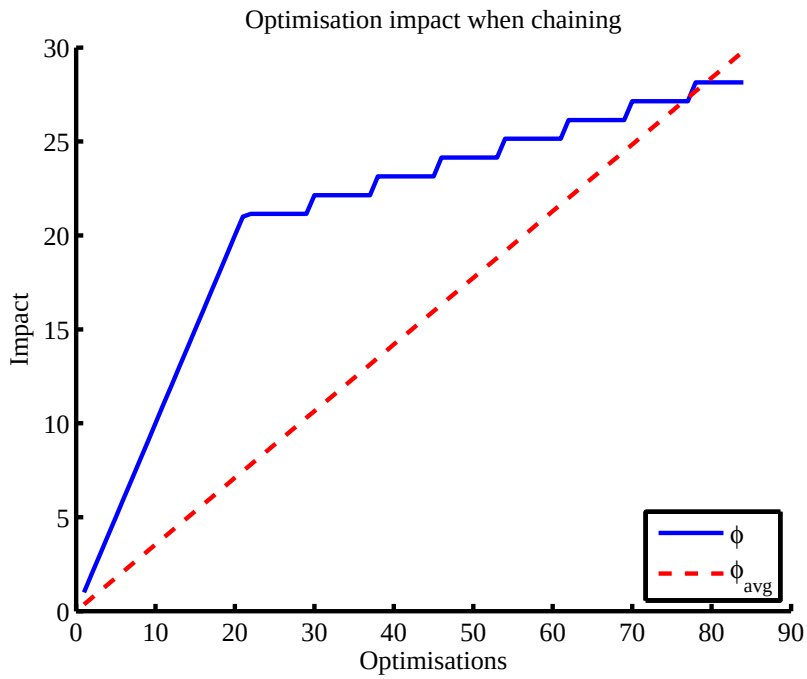


Fig. B.7: This figure shows the relation between the number of optimisations and the corresponding impact when performing chaining optimisations. The solid line shows the impact when working with the transformed graph. In the beginning every optimisation is performed in the critical path, where every optimisation results in a reduction. When the average length of the off critical path paths is reached, chaining optimisations need to be performed in every path to induce a reduction. The dashed line denotes the average impact of the chaining optimisations.

$$M = \left(1 - \frac{\gamma - 1}{P(NOO_{PAR}) - 1}\right) CP \tag{B.13}$$

when the critical path is changed so that the critical path has the same length as the average of the off critical paths, the scenario changes to the highly distribution extreme. The rest of the chaining optimisations are handled as so.

Since most algorithms do not fall into one of the two extremes, when estimating the impact of a certain number of chaining optimisations the average measure is more representative than the measure obtained based on considering the two extremes.

Furthermore, it can be shown that the real impact from optimisation will be equal to or better than the average of the simplified model[1]. Using such a measure will therefore ensure that the estimates of the chaining optimisation impact are not overestimated.

The measure for the impact of a chaining optimisation can therefore be denoted by the following:

$$\varphi(NOO_{PAR}) = \left(\frac{M}{NOP} + \frac{CP - M}{NOP - M}\right) \tag{B.14}$$

With all these defined we are now ready to find the number of optimisations need to fulfil the real-time constraint, and define the metric expressing how hard it is to reach this implementation. We call this metric for implementation hardness.

4.2 Implementation hardness

Knowing the maximal number of possible optimisations and the minimum needed to meet the time constraint, the ratio (Eq: B.15) between these two indicates how much the implementation needs to be investigated. Using the analogy with the implementation tracks, a number close to one indicates that almost all possible optimisations in the algorithm need to be considered, and only a very limited number of tracks will lead to a solution. Finding these solutions will require a lot of effort. On the other hand a number close to zero indicates that few optimisations are required

[1] equal to or better than only applies when considering the lower integer value of the achieved reduction, since no partial operators exist.

to meet the time constraint and thus almost every implementation track will result in a satisfying solution. Hereby less effort is probably needed.

$$\tau(t_c) = \frac{|NOO_{PAR,Chain}(t_c)|_{min}}{|NOO_{PAR,Chain}|_{max}} \tag{B.15}$$

Using $\tau(t_c)$ in Eq. B.2 we are now able to express the implementation hardness, $\tau(t_c)$, and thereby refine the estimated implementation effort.

5 Results

Similar to when we developed the implementation effort estimation technique in [3], we will verify the proposed improvement by first building a model on basis of the same training data as in [3], and then validate the model with a set of validation data, which is also the same as used earlier. By doing so, we are able to first test if our considerations are valid and tune the proposal, and still use the second set of real-life data to evaluate whether it generalizes.

To summarize, the training data originates from two different application types that are both developed as academic projects in universities in France. The training data has therefore not been produced specifically for this project, but is comparable to data from industrial projects. The first application is composed of five different video processing algorithms that are able to track moving objects in a video sequence. The second application is a cryptographic system, where we use the hashing algorithms, MD5, AES and SHA-1, as well as a combined crypto engine, which is also part of the system. The developers for the training data have been a Ph.D. student and M.Sc.EE. students, as can be seen in Table B.1.

The validation data originates from a local company, ETI A/S, which is a Danish SME. The dataset contains algorithms from a state-of-the-art network system and consists of Ethernet applications implemented on FPGAs, as well as corresponding testbeds. The system is a real-time system with hard time constraints, and all algorithms were implemented as to meet these constraints. The developers for the validation data had some experience before starting the implementation as shown in Table B.1. For more information about the data we would like to point the reader to [3].

5.1 Training Data

Fig. B.8 shows the training data where the uncorrected complexity as defined by the number of linearly independent paths (as defined in [3]) is plotted in relation to the needed effort. A small update in the method of how to measure the independent paths have been applied compared to [3]. This implies that we now only measure the core of the algorithm, which is the part going to be implemented on the FPGA, and do not include small fragments of data formatting code. Taking these data and applying the original experience transformation on the data, results in the picture shown in Fig. B.9. A least-squares fit trend line can be extracted to form our model (Eq. B.1):

$$Effort = A(\eta(Dev) \cdot P(alg))^b \tag{B.16}$$

where the trim parameters $A = 0.196$ and $b = 1.191$. This is depicted as the dash-dot-dash line.

In Fig B.11, the new parameter $\tau(t_c)$ taking the real-time constraint into account is applied. The $\tau(t_c)$ value for the different algorithms is shown in the upper part of Table B.2. This changes the complexity for the different algorithms a little, and a new least-squares fit line of our model is depicted with the dashed line. The trim parameters of our model (Eq. B.2):

$$Effort = A(\eta(Dev) \cdot P(alg) \cdot \tau(t_c))^b \tag{B.17}$$

are now $A = 0.209$ and $b = 1.181$.

A comparison of the two models is shown in Table B.3, where the model taking the real-time constraint into account performs slightly better. However the result is not statistical significant. However, we continue with the

5.2 Validation Data

We continuing by validating the correctness of the model using the validation data. In Fig B.11, the corrected validation data are shown together with the model, which is depicted by the dashed line, and a 95% confidence interval. Both the model and confidence interval are extracted from the training data. It is clear that most algorithms fit nicely with

the proposed model and are well within the confidence interval. The exceptions are algorithm SS4 and SS5. In the next section we will discuss this in details. The mean and variance of the prediction errors are shown both with and without SS4 and SS5 in Table B.4.

5.3 Discussion of result

Most of the algorithms fit nicely with the proposed model and are well within the confidence interval. The improved model indeed does perform better than the original model, which had a mean error of 0.2 and a variance of 8. Taking a closer look at Table B.2 shows that for all the Ethernet algorithms, except for SS3, we obtain an implementation hardness value $\tau(t_c)$ very close to one. This indicates that the implementations have been very close to the maximum achievable with the algorithm. This fits very well with the knowledge that these algorithms are used in state of the art high performance systems. However, the result for SS3 shows that it should have been possible to choose a less optimised solution and still meet the constraints, which would have resulted in a reduction of the implementation time, at least if the model holds for this algorithm.

An exception to these results are the SS4 and SS5 algorithms, where the estimates do not fit the model. Looking at table B.2 again, we see that their implementation hardness value is set to 1. This is an error value actually indicating that the time constraint cannot be met with the current algorithm, and an algorithm transformation is needed. This indicates that the algorithm which is used for estimating the complexity of the implementation and thereby the needed effort, will not be able to fulfill the requirements, and an algorithm transformation is probably needed. Not going into details with the two algorithms, we can say that their final implementations involve a lot of bit manipulation which is not easily reflected in the initial C algorithm which is used for the measurement. A safe conclusion is therefore that if the implementation hardness factor indicates the need for algorithm transformation, the result would hardly be covered by the proposed model.

Furthermore it is also important to stress that our set of data originates from a single company with few developers. So strictly speaking we can only conclude that this model can be applied to the specific SME setup involved in the study and partially to the academic environment

studied. A large volume and variety of experimental data for training and validation is needed to generalise our model. Also, the model can be refined with more parameters for more precise results.

Table B.1: Facts about the developers. Developers for training data (Top) and validation data (Bottom)

Developer	Education	Years in the domain
Dev 1	Ph.D. stud.	0
Dev 2	Stud. (EE)	0
Dev 3	Stud. (EE)	0
Dev 4	Stud. (EE)	0
Dev 5	BSc.EE.	9
Dev 6	MSc.EE.	15
Dev 7	MSc.EE.	9
Dev 8	MSc.EE.	8
Dev 9	MSc.EE.	8

6 Conclusion

Accurate development time estimates are an essential tool for project managers working in the embedded systems industry. Obtaining such estimates is challenging and in particular very few existing works can provide hardware implementation effort estimates. In this paper we have presented our contribution to this topic, namely a systematic and structured approach for estimating the hardware implementation effort of hard real-time constrained applications.

The underlying idea off this work is that implementing a system is more difficult when hard real-time constraints must be fulfilled since designers have to identify a or the suitable implementational track(s) that lead to a satisfying solution. We have proposed an optimization model which takes some of the most common optimization techniques into account as well as the order in which they are applied.

In particular we have suggested a set of two metrics which are used to characterise the effects of optimisations. The first one, the implementa-

Table B.2: Training Data (Top) and Validation Data (Bottom). Algorithms are related to the implementation hardness, $\tau(t_c)$, the developers, and their experience at the given time. Complexity is not corrected.

Algorithm	Complexity	$\tau(t_c)$	Developer	Dev. Exp.
T1	10	0.97	Dev 1	2
T2	24	0.99	Dev 1	10
T3	12	0.91	Dev 1	18
T4	14	0.96	Dev 2	1
T5	4	0.89	Dev 1	20
MD5	10	0.98	Dev 3	1
AES	10	0.99	Dev 4	8
SHA-1	27	0.98	Dev 4	14
Combined	59	0.99	Dev 4	14
SS1	25	0.99	Dev 6,7	150
SS2	35	0.98	Dev 5	150
SS3	17	0.26	Dev 5,6,7,8	150
SS4	50	1	Dev 6	6
SS5	29	1	Dev 7	3
SS6	25	0.99	Dev 5,6,7	3
Ethernet app	60	0.99	Dev 5,6,7,8,9	150
App 4	9	0.94	Dev 6	150

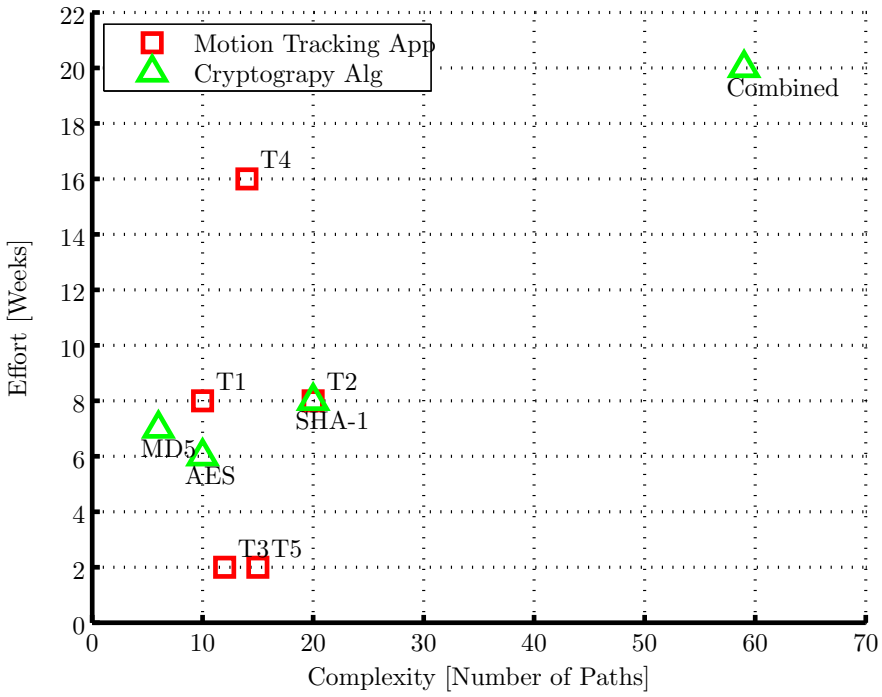


Fig. B.8: Relation between the implementation effort [number of weeks] and the uncorrected complexity.

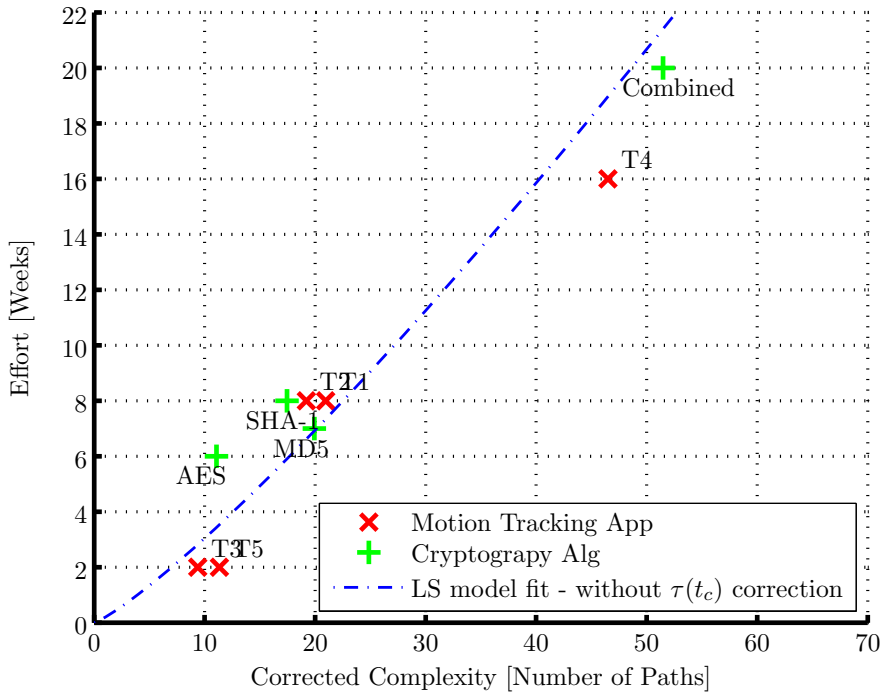


Fig. B.9: Relation between the implementation effort [number of weeks] and the complexity corrected according to the designersŠ experience model.

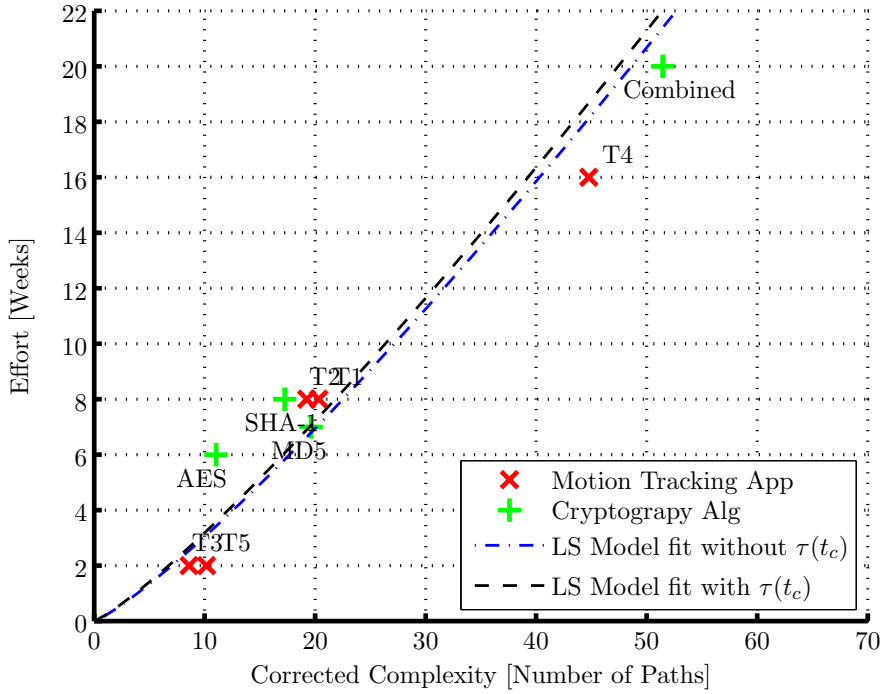


Fig. B.10: Relation between the implementation effort [number of weeks] and the complexity corrected according to the designersŠ experience model and the hardness of meeting the real-time constraint.

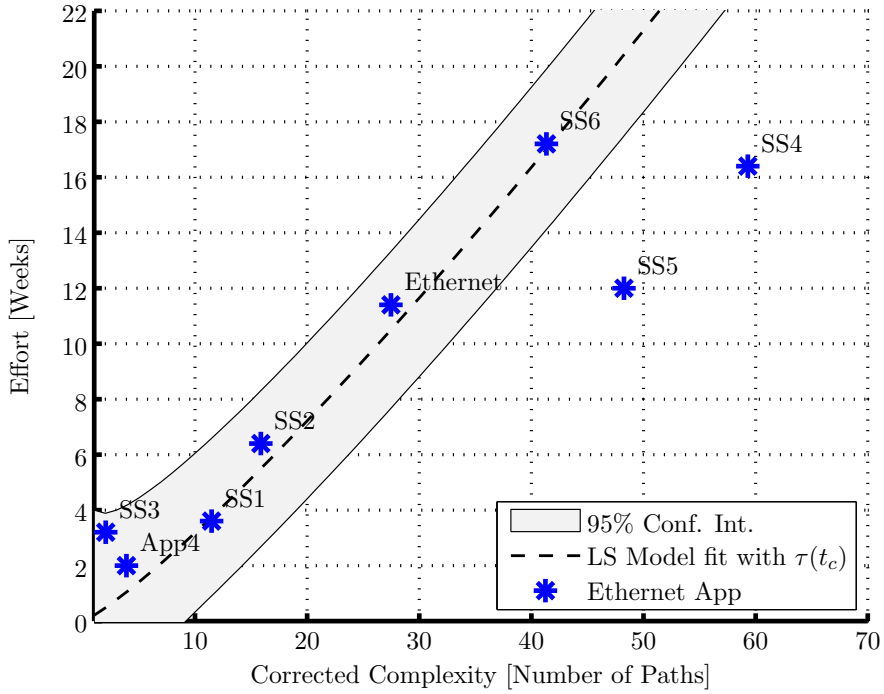


Fig. B.11: Validation data plot: relation between implementation effort [number of weeks] and complexity, corrected according to the designersŠ experience model and the hardness of meeting the real-time constraint.

Table B.3: Comparison of the development time and estimated development time for the two models measured in weeks.

Algorithm	Original Model Error	New Model Error
T1	0.67	2.19
T2	1.38	3.29
T3	-0.82	-1.56
T4	-2.96	-4.45
T5	-1.53	-3.14
MD5	0.09	0.51
AES	2.57	6.65
SHA-1	2.10	5.28
Combined	-1.40	-2.83
Mean (Variance):	1.50 (3.39)	1.42 (3.07)

Table B.4: Comparison of the development time and predicted development time measured in weeks.

Algorithm	Estimation Error
SS1	-0.14
SS2	0.91
SS3	2.71
SS4	-9.64
SS5	-8.42
SS6	0.19
Ethernet app	0.90
App 4	0.96
Mean (Variance):	-1.56 (22.02)
Mean without SS4 and SS5 (Variance):	0.92 (0.98)

tion hardness metric, reflects how hard it is to reach an implementation satisfying the real-time constraints for the application. The second one, the parallelism distribution metric, reflects how the distribution of parallelism in an algorithm influences the impact of the optimisations.

The experimental results is not unambiguous: for the model the major improvement of the accuracy comes from refining the way the complexity of training data is measured compared to our prior work. A small and not statistical significant improvement comes applying the implementation hardness measure. When validating the model with the validation data, most of the data approve the model, and fit with it very well. A mean error of 0.92 week (variance 0.98) is achieved, when not considering two outlying data points for which our implementation hardness measure indicate that the time constraint for these algorithms can not me met. Strong algorithm transformation is probably need here and safe conclusion will therefore be that the proposed model will hardly cover these cases.

In order to strengthen the result this work need to be evaluated with more cases. The work would also benefit from making room for other optimisation strategies such as pipelining.

References

[1] R. Nass, D. Blaza, and M. Barr, "2009 embedded market study," http://www.techonline.com/learning/livewebinar/216500641.

[2] B. W. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece, *Software Cost Estimation with COCOMO II*. Prentice Hall, 2000.

[3] R. Abildgren, J. Philippe Diguet, P. Bomel, G. Gogniat, P. Koch, and Y. Le Moullec, "A priori implementation effort estimation for hardware design based on independent path analysis," *EURASIP Journal on Embedded Systems*, September 2008.

[4] A. J. Albrecht, "Measuring application development productivity," in *Proc. IBM Applications Development Symp.*, 1979.

[5] T. Jones, *Programming Productivity*. New York: McGraw–Hill, 1986.

[6] D. Jeffery, G. Low, and M. Barnes, "A comparison of function point counting techniques," *Software Engineering, IEEE Transactions on*, vol. 19, no. 5, pp. 529–532, May 1993.

[7] W. Fornaciari, F. Salice, U. Bondi, and E. Magini, "Development cost and size estimation starting from high-level specifications," in *Proceedings of the ninth international symposium on Hardware/software codesign*, 2001, pp. 86–91.

[8] A. Agarwal and R. Shankar, "Cost feasibility analysis for embedded system development and the impact of various methodologies on product development cycle," Florida Atlantic University, Tech. Rep., 2008.

[9] L. Piga and S. Rigo, "Comparing rtl and high-level synthesis methodologies in the design of a theora video decoder ip core," in *Programmable Logic, 2009. SPL. 5th Southern Conference on*, 2009.

[10] Y. Le Moullec, N. B. Amor, J.-P. Diguet, M. Abid, and J.-L. Philippe, "Multi-granularity metrics for the era of strongly personalized SOCs," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2003.

Paper C

Algorithm-Architecture Affinity – Parallelism Changes the Picture

Rasmus Abildgren, Aleksandras Šaramentovas, Paulius Ruzgys,
Peter Koch, and Yannick Le Moullec

This paper has been published in
Proceedings of the Design and Architectures for Signal and Image Processing Workshop,
2007.

The layout has been revised.

Abstract

Reducing the time-to-market factor is a challenge for many embedded systems designers. In that respect, hardware-software partitioning is a key issue which has been studied during the last two decades. In this paper we present an extension to recent works dealing with metrics for guiding the hardware-software partitioning step. This extension builds upon and complement our own work with metrics in the Design Trotter project, and is combined with the affinity metric approach. We show that the proposed extension improves the original affinity metric in terms of parallelism detection, and thus can help system designers to make wiser hardware-software partitioning decisions, which in turn reduces the time-to-market factor.

1 Introduction

In order to achieve more advanced and faster services in embedded systems, increasingly sophisticated algorithms are used. To keep abreast with the increased need for processing power, heterogeneous multiprocessor platforms are introduced, which includes GPPs, DSPs and FPGAs.

Introducing this variety of processing elements (PEs), not only increases the computational capacity of embedded systems but also adds various computational properties. To exploit this increased capacity and properties, the designer needs to find the best suited PEs for the different system functionalities. By considering these facts together with all the system constraints (Area, Time, Power, Price, Development Time), it becomes a non-trivial task to decide how the system functionality should be mapped on the architecture.

To handle this task system level design methodologies have been developed, including structured design space exploration (DSE). A suite of academic DSE frameworks, e.g. [1–3], as well as commercial tools have been proposed, in order to provide the design engineer with qualitative information for partitioning.

Exploring the design space with optimising for different constraints is known to be $\mathcal{NP}$ hard [4]. The DSE in these frameworks is therefore carried out as heuristic simulations, which still can be a time-consuming but necessary task for state-of-the-art large scale products. Large companies

can usually find these resources and keep up with their competitors.

However, small and medium enterprises (SMEs), which typically sell state-of-the-art products of much smaller volumes, must also stay on the competitive edge. They are also restricted by the time-to-market factor, and can also benefit from using system level design methodologies (SLD) and tools. Unfortunately, many SMEs can not afford tools and specialists like big companies, and therefore have problems with changing their design methodology into SLD methodology.

We have examined the design methodology of a high-tech company in Denmark and found that the design space exploration phase in their overall design trajectory is limited in the sense that their partitioning depends on prior design, designers intuition and experience, and in rare cases on ad hoc analysis. Danish Technological Institute, a consulting company helping many SMEs incorporating new research results, agrees on that picture in most SMEs [5].

As a consequence of sticking to ad hoc design methodologies, SMEs development often run into situations where redesigning part of the system is necessary and therefore increases the time-to-market.

In this paper we propose an extension to the existing affinity metric proposed in [6] for guiding the partitioning of the system specification, and help making the DSE faster and easier. The rest of the paper is organised as follows. In section 2, the existing affinity metric is presented and examples for the need of an extension to the original metric are shown. In section 3 the new proposed metric for parallelism is presented. The benefits of the proposed parallelism metric are illustrated in section 4 by means of a Reed-Solomon decoder case-study. Finally we conclude in section 5.

2 Affinity Metric

This section summarises the affinity metric proposed by D. Sciuto et.al. in [6, 7], and argues for the need of an extension of this metric. The affinity metric is designed to guide the design partitioning of system specification between general purpose processors, DSP processor, and FPGA/ASIC. The metric consists of a triplet of values $(A_{GPP}, A_{DSP}, A_{FPGA})$ indicating the match between the processing elements and the examined code. The individual values in the metric are calculated based on 14 other metrics

which are designed to measure the source code for certain patterns highly correlated with architectural properties. The measurement is a static analysis of the source code and the metrics are defined as ratios between lines with specific properties, e.g., the ratio between lines with a condition and the total number of lines, or defined as the number of assignment of a special type related to the total number of assignments. The metrics measure properties such as data types, Harvard architecture patterns, MAC patterns, and bit manipulation.

To illustrate how the affinity metric works on a real life example, we have applied it onto c-code (Fig C.3) calculating a matrix multiplication. The results of the different metrics are shown in table C.1:

Table C.1: The affinity value for the matrix multiplication algorithm, where A_{xxxx} indicates the match between the processing element type and the code. 0 =no matching, 1 =perfect match.

A_{GPP}	A_{DSP}	A_{FPGA}
0.89	0.96	0.39

The normalised metric values indicate that the best architecture matching the algortihm is a DSP architecture, which the designer could easily rely on. An in-depth analysis of the code shows that besides the already extracted properties from the affinity metric, a high degree of inherent parallelism is present in the matrix multiplication algorithm. This is further discussed in section 3. A high degree of inherent parallelism indicates that the algorithm is suited for parallel execution. This is one property of a FPGA architecture, and the original affinity metric does not consider it.

3 Parallelism metric

From the analysis of the matrix multiplication shown in Fig C.3, we see that the inherent parallelism of an algorithm is an important parameter. Therefore it would be beneficial to measure the degree of inherent parallelism in the algorithm and use this in calculating the A_{FPGA} value of

the affinity metric.

One of the first metrics considering the parallelism is Amdahl's speedup metric [8]. Here the potential execution speedup of an algorithm is defined as the ratio between the sequential execution, and the fully parallelised execution. What determines the fully parallelised execution is the critical path in the algorithm.

This is also the case for more recent parallelism metrics e.g. [9, 10], so let us consider the critical path by looking at precedence graphs.

Definition 1. *Let $G = (N, E)$ represent the precedence graph of a method, m, where N represents the set of nodes n_i and E is the set of edges $e_{i,j}$. A node n_i can have a source node and a destination node. If the node does not have a source node, it is defined as a start node, and if the node does not have a destination, it is a sink node. If a dependency between two nodes; the parent node, n_i and the child node, n_j, exists, it is connected with an edge $e_{i,j}$. The node, n_j, cannot execute before it has obtained data from its parent(s).*

Using definition 1, we can now express the critical path of algorithm using the following definition:

Definition 2. *The critical path, CP, is a set of nodes $n_{start}, \ldots, n_i, \ldots$ $\ldots, n_{sink}$ and associated edges $e_{start,h}, \ldots, e_{i,j}, \ldots, e_{k,sink}$ forming a path, p, from a start node, n_{start}, to a sink node, n_{sink}, for which the sum of costs are a maximum:*

$$CP = \max cost(\{n_{start}, e_{start,h}, n_h, \ldots, n_i, e_{i,j}, n_j, \ldots \\ \ldots, n_k, e_{k,sink}, n_{sink}\}) \quad \text{(C.1)}$$

A way to measure the inherent parallelism that uses the critical path is the γ metric developed in our previous work [9] which is defined as:

$$\gamma = \frac{NOP}{CP} \quad \text{(C.2)}$$

where we consider the nodes to be atomic, meaning that NOP represents the total number of operations in the precedence graph.

The metric described in (C.2) expresses the level of inherent parallelism of the algorithm by calculating the ratio between the number of operations in the algorithm, and the number of operations in the critical

path. In this case, where we consider all nodes as basic operations, NOP is equivalent with the total number of nodes N. This metric is organised such that with no inherent parallelism its gives the value 1. The metric value increases along with the inherent parallelism.

The affinity metric [7] on the other hand is in comparison a normalised measure, where zero indicates the worst match and one indicates a perfect match between the algorithm and the architectural property. Using the γ for expressing the inherent parallelism will lead to non-comparable results. A metric expressing the parallelism together with the affinity metric should have the same normalised properties. To suit these properties we can rewrite the γ metric into a normalised metric:

$$\gamma' = 1 - \frac{CP}{NOP} \tag{C.3}$$

The affinity metric is based on textural analysis of the source code and therefore does not refer to the number of operations, critical path or any of the terms used above for γ and γ'. Instead it operates with source lines which contain certain patterns.

In order to cope with the parallelism measure inside this source line based framework, we propose a new metric, θ, inspired by the γ' metric. θ is defined as:

$$\theta = 1 - \frac{S_{CP}}{S_m} \tag{C.4}$$

where S_{CP} is the number of source lines included in the critical path and S_m is the total number of source lines in the code. To emphasise the weight of the critical path, a loop unrolling is need to be performed before measuring S_m and S_{CP} of the θ metric.

This way of expressing the parallelism is not equivalent with γ' since every source line in a high level language will usually lead to more than one atomic operation. The danger is that the number of atomic operations highly depends on the programmers coding style. A compact code will result in more operations per source line than a fragmented code with many intermediated/temporary variables which come close to one operation per code line. It is therefore impossible to obtain the same precision, as the modified and normalised γ' metric.

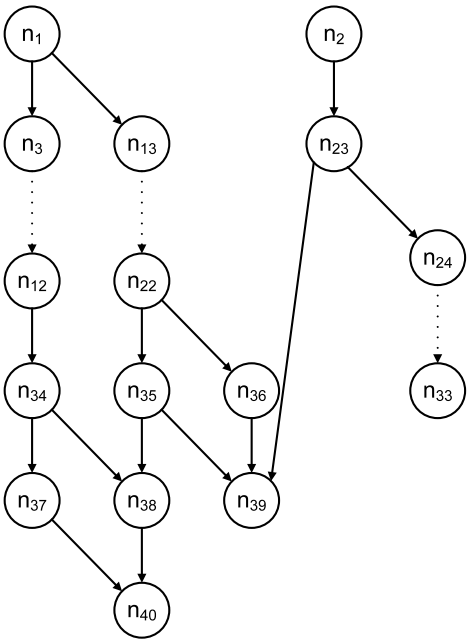


Fig. C.1: Precedence graph of random 1 algorithm.

To examine their differences, extreme cases, i.e. a purely sequential and a fully parallel execution as well as two random cases have been considered. The two random execution graphs are shown in Fig C.1 and Fig C.2. Comparing the γ' metric and the θ metric on these cases provides us with the results shown in the four first lines of table C.2. We here consider $N = 40$ in the precedence graphs, where a source line on average corresponds to four nodes. The sequential execution gives, as expected, the same result for both metrics i.e., 0. The fully parallel execution however, gives a slightly different result for the two metrics, $\gamma' = 0.975$ and $\theta = 0.9$. None of them reach the value 1 for a full parallel execution, because of the way CP is defined. But we notice that θ gives a lower score than the γ' metric. This is due to the smaller number of code lines compared with the number of nodes, which influences the ratio. For the random case there are larger differences (0.65 vs. 0.56) and (0.7 vs. 0.75).

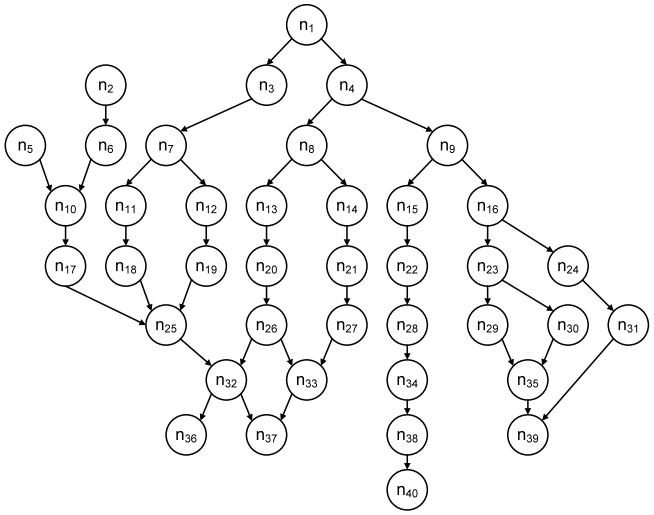


Fig. C.2: Precedence graph of random 2 algorithm.

Table C.2: Differences between the γ' and θ metric.

	γ'	θ
Sequential:	0	0
Parallel:	0.975	0.9
Random 1	0.65	0.56
Random 2	0.7	0.75
Matrix Multiplication:	0.999	0.989

Table C.3: The original affinity metric values for GPP, DSP, and FPGA and the proposed metric (FPGA&θ) for the Reed-Solomon Decoder algorithm. The performance (latency) of the different architectures are also shown.

	GPP	DSP	$FPGA$	$FPGA\&\theta$
Affinity	0.717	0.795	0.205	0.806
Latency $[\mu s]$	-	514	2278	244

Even though the θ metric and the γ' metric do not give similar results, θ still gives a good indication of the algorithms affinity to a parallel architecture. Let us discuss this issue by re-considering the matrix multiplication case given by:

$$\mathbf{C} = \mathbf{AB} \tag{C.5}$$

where $\mathbf{C} \in \mathbb{R}^{X \times Z}$, $\mathbf{A} \in \mathbb{R}^{X \times Y}$, $\mathbf{B} \in \mathbb{R}^{Y \times Z}$ are matrixes where X, Y, Z denotes the dimensions. Here the dimensions are $X = Y = Z = 10$. The c-code taken from the DSPstone project [11] is shown in Fig C.3, and we see that the kernel of the algorithm consists of multiplications, memory reads and writes together with some indexing controls. A precedence graph of the kernel of the algorithm is shown in Fig C.4. The results of the examination of the algorithm with the two metrics are also shown in table C.2. From this we see that there is an insignificant difference between the two metrics (i.e., 0.999 and 0.989), which is due to the high number of nodes and unrolled source lines. From these cases it appears that the newly proposed metric θ serves its purpose of indicating parallelism.

4 Case study

In this section we present a case study, which expresses the benefits of the introduced metric, before selecting the architecture for a Reed-Solomon decoder.

```
int matrixMul(static int A[X*Y],
              static int B[Y*Z],
              static int C[X*Z])
{
  int *p_a = &A[0] ;
  int *p_b = &B[0] ;
  int *p_c = &C[0] ;

  int f ;
  int i ;
  int k ;

  for (k = 0 ; k < Z ; k++)
    {
      p_a = &A[0] ;  /* point to the beginning of array A */

      for (i = 0 ; i < X; i++)
         {
           p_b = &B[k*Y] ;    /* take next column */
           *p_c =  0 ;

           for (f = 0 ; f < Y; f++) /* do multiply */
              *p_c += *p_a++ * *p_b++ ;

           *p_c++ ;
         }
    }
  return(&C[0]) ;
}
```

Fig. C.3: Matrix multiplication example.

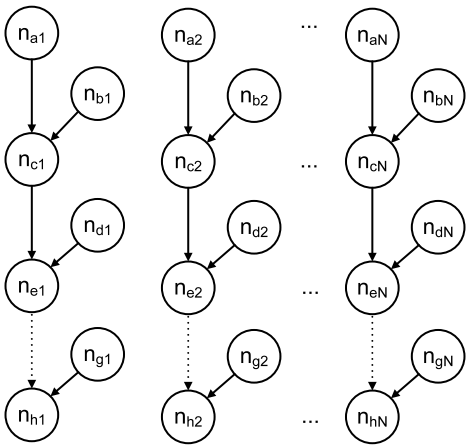


Fig. C.4: Precedence graph of the kernel of the matrix multiplication example.

4.1 Reed-Solomon Decoder

Reed-Solomon codes are a forward error correction codes used in many modern communication systems. The decoder is able to detect and correct some bit errors which have occurred doing the transmission. It is an algorithm which involves many conditional branches in order to detect and repair errors.

The algorithm has been examined with the affinity metric, and the results are shown in table C.3. The table shows the original affinity metric values for GPP, DSP and FPGA architectures and the affinity metric for FPGA with our new extension (added as an extra parameter for FPGA metric before normalisation as in [7]). We see that the Reed-Solomon decoder has the highest score (0.795) on a DSP architecture with the original affinity metric, however, the score for FPGA architecture increases significantly (from 0.205 to 0.806) when including our extension, and thereby gets the highest score. To verify the results, the algorithm has been implemented on a Analog Devices TigerSHARK ADSP-TS201 DSP and a Xilinx Virtex II FPGA, in high-level languages (C and Handel-C, respectively). The latency for decoding one block was measured on both platforms. The FPGA implementation was done in two steps: first, a version without exploiting the parallelism, which corresponds to the

original affinity metric interpretation, and second, a version exploiting the inherent parallelism. These latencies are also shown in table C.3.

Inspecting the results shows that the best performance is obtained by the parallelised FPGA implementation, with a latency of $244\mu s$. We can then deduce that using the original affinity value for FPGA in this case will not disclose the architectures potential for the Reed-Solomon algorithm. Without considering the parallelism, the designer would make an inefficient partitioning choice.

Using the extended metric that we propose gives a better indication of the affinity between algorithm and FPGA architecture, thus helps the designer to make wiser partitioning decisions.

5 Conclusion

In this paper we have proposed an extention of the affinity metric [6], in order to improve the capability to measure the algorithm-architecture affinity for FPGA. The extension consists of a new metric derived from some of our previous work [9]. This new metric provides a mean for measuring the inherent parallelism of the algorithm inside the source code. We have shown that adding this new metric to the original affinity metric improves its score for FPGA matching.

References

[1] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, and H. Zheng, "Overview of the ptolemy project," Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720, Technical Memorandum UCB/ERL M03/25, July 2003.

[2] C. Erbas, A. D. Pimentel, M. Thompson, and S. Polstra, "A framework for system-level modeling and simulation of embedded systems architectures," *EURASIP Journal on Embedded Systems*, 2007.

[3] J. Riihimäki, P. Kukkala, T. Kangas, and M. H. T. D. Hämäläinen, "Interfacing uml 2.0 for multiprocessor system-on-chip design

flow," in *Proceedings of International Symposium on System-on-Chip*, November 2005, pp. 108 – 111.

[4] Z. A. Mann and A. Orbán, "Optimization problems in system-level synthesis," in *Proceedings of the 3rd Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications*, 2003. [Online]. Available: citeseer.ist.psu.edu/mann03optimization.html

[5] T. S. Olesen, "Private conversation about Danish SMEs design methodologies," August 2006.

[6] C. Brandolese, W. Fornaciari, L. Pomante, F. Salice, and D. Sciuto, "Affinity-driven system design exploration for heterogeneous multiprocessor soc," *Computers, IEEE Transactions on*, vol. 55, no. 5, pp. 508–519, May 2006.

[7] D. Sciuto, F. Salice, L. Pomante, and W. Fornaciari, "Metrics for design space exploration of heterogeneous multiprocessor embedded systems," in *Hardware/Software Codesign, 2002. CODES 2002. Proceedings of the Tenth International Symposium on*, 6-8 May 2002, pp. 55–60.

[8] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS spring joint computer conference*, 1967.

[9] Y. L. Moullec, N. B. Amor, J.-P. Diguet, M. Abid, and J.-L. Philippe, "Multi-granularity metrics for the era of strongly personalized SOCs," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2003.

[10] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for internocnection-constrained heterogenous processor architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 4, 1993.

[11] "DSP Stone," 1995, http://www.ert.rwth-aachen.de/Projekte/Tools/DSPSTONE/dspstone.html.

Paper D

HSDPA Design Space Exploration and Implementation Guidance with Design-Trotter

Aleksandras Šaramentovas, Paulius Ruzgys, Rasmus Abildgren,
and Yannick Le Moullec

This paper has been published in
Proceedings of the 6th IEEE International Conference on Information, Communications and Signal Processing, pp. 1–5, 2007.

The layout has been revised.

Abstract

This work addresses some of the implementation challenges for recent and future wireless communication systems. More specifically this paper describes a design methodology for design space exploration and implementation guidance and illustrates its practical usage and benefits by applying it to two sub-parts (i.e., the turbo-encoder and turbo-decoder) of the HSDPA technique. The implementation examples and results show how the proposed methodology based on our tool "Design-Trotter" can guide system designers in selecting and/or building the most appropriate architecture for their application.

1 Introduction

In order to satisfy the ever-increasing need for data-traffic and high-speed services in the wireless domain, new techniques have been developed to increase the spectral efficiency of current third generation systems to support high user data rates. The High Speed Downlink Packet Access (HSDPA) technique [1], which has been validated by the Third Generation Partnership Project (3GPP) in the specification of the Release 5 is a significant step to boost the WCDMA performance for downlink packet traffic, enabling user peak data rates up to 14 Mbps. HSDPA offers new opportunities for wireless communications but also raises a number of implementation challenges such as platform selection (e.g., DSP-processor, FPGA, GPP) and optimizations (e.g., time, power and area).

In order to alleviate system designers from time-consuming and error-prone design tasks which increase the time-to-market factor, a systematic and efficient design methodology is highly desirable. This work illustrates how designers can benefit from a guidance methodology for the implementation of the HSDPA technique by means of design space exploration (DSE) with the Design-Trotter tool [2]. In particular we investigate how DSE combined with the characterization of the application by means of metrics provide a design trajectory allowing early and right decisions for selecting and/or building the most appropriate target architecture according to the system specifications, and thus reducing the time-to-market factor.

The remainder of the paper is organized as follows: Section 2 intro-

duces the main steps of the proposed design methodology and discusses some of the key issues related to design space exploration and how to move from high-level estimates to an actual implementation. Then section 3 illustrates how the proposed methodology can be used for guiding the implementation of some of the most critical sub-parts of the HSDPA concept (i.e., the turbo-encoder and turbo-decoder) by means of design space exploration examples and implementation results. Finally we conclude in section 4.

2 Methodology

Modern system development requires a design methodology which enables the designer to explore different implementations in order to choose one which fulfils the performance requirements for the product. In this work we consider a design methodology build upon the design space exploration tool Design-Trotter. The overall design methodology is summarised in Fig.D.1 and the main steps are presented hereafter. Further details about the tool Design-Trotter can be found in [2].

The task of analysing an algorithm with a design space exploration tool consists of characterizing the algorithm, in such a way that the designer is able to get useful information about the performance of different implementations, typically in terms of a resource vs. execution time, or area vs. time curve.

Based on the design space exploration results, a particular solution can then be further explored and implemented. Some tools try to do this automatically based on the algorithm description, however this task is still done more or less manually in many situations. Due to the time-to-market parameter, high-level languages are increasingly used for implementation. It is therefore important that the used high-level languages are able to express the detailed needs in order to implement the chosen design.

In the following we describe the individual tasks and illustrate some of the issues involved when going from DSE estimates to real implementations using high-level languages.

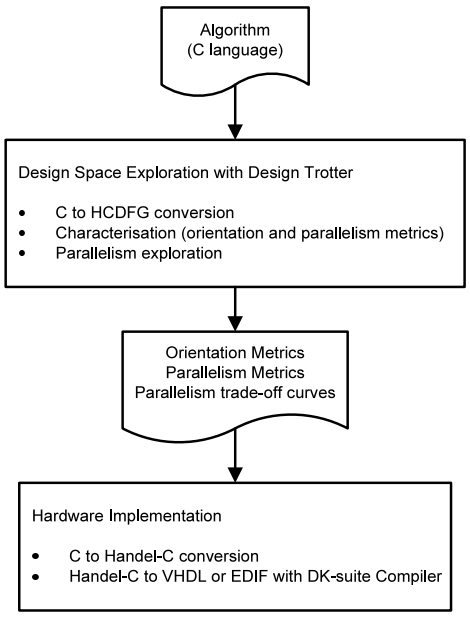


Fig. D.1: Our mehodology for rapid development of wireless applications.

2.1 Design Space Exploration with Design-Trotter

Design-Trotter [2] is an academic design space exploration tool conjointly developed by Laboratoire d' Électronique des Systémes TEmps Réels (LESTER), Université de Bretagne Sud, France and Center for Software Defined Radio (CSDR), Aalborg University, Denmark.

For guidance purposes metrics are computed to rapidly stress the proper architecture style for the application, e.g., the ratio of explicit parallelism versus the pipeline depth, the need for complex control structures, the requirements in terms of local memories and specific bandwidth, and the need for processing resources for specific computations or address generation. Design-Trotter computes three orientation metrics [3]: the Memory Orientation Metric (MOM), the Control Orientation Metric (COM), and the criticity (average potential parallelism) of a function (γ).

Since parallelism has a direct impact on several performance factors such as execution time, energy consumption, area, etc., Design-Trotter explores the potential parallelism of an application in terms of i) type (data-transfer and data processing), ii) granularity level, and iii) type

(spatial, temporal). Design-Trotter rapidly provide dynamic exploration of an application by means of parallelism vs. delay trade-off curves on which a point corresponds to a potential architecture.

The analysis of the algorithm under consideration in Design-Trotter is done automatically, and provide the designer with the above mentioned information. The designer use these information to identify which possible solutions in the solution space are fulfilling the requirements. Since the solutions provided by Design-Trotter are estimates, it is important that these estimates are close to the performance of the real implementation.

2.2 Design-Trotter Solution to Handel-C

Having the algorithm and the design suggestions provided by Design-Trotter, the next task is to perform the actual implementation. To keep the development time short, high-level languages like Handel-C [4] are preferred.

Each solution proposed by Design-Trotter could be implemented on an FPGA using a HDL. Ideally this could also be the case for high-level languages, however, the main problem at this point is how to achieve this precision. To settle it the following elements are used: the resource schedule details provided by Desing-Trotter and the ”par” statement in Handel-C. Firstly the C source code used in Design-Trotter is converted to Handel-C; secondly by referring to the schedule details of the desired Design-Trotter solution, we can manually express parallelism inside the top-level blocks of an algorithm using the ”par” construct in the corresponding Handel-C code parts; thirdly the Handel-C code in DK-Suite [4] is compiled to an EDIF design file, used for further implementation on the FPGA.

The structure of the compiled Handel-C code depends on the assignments in the Handel-C code, meaning that every assignment in the Handel-C code have a corresponding circuit which takes a clock cycle to execute.

Due to this fact, one can imagine that the Design-Trotter result in terms of cycle-budget will differs from the corresponding Handel-C result.

As an example, let’s depict the schedule details of the encoder’s interleaver block, shown in Fig D.3. The C source code of this interleaver is shown below:

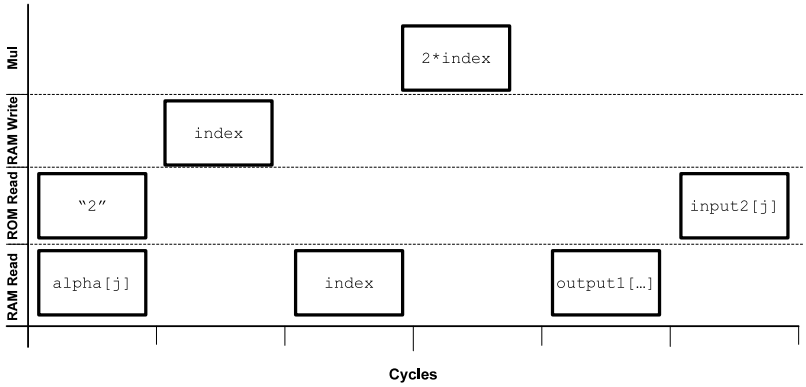


Fig. D.2: Schedule details of the interleaver, derived from Design-Trotter.

```
/* interleave output of upper RSC encoder */
for(j=0; j<FrameLength; j++){
  index = alpha[j]; //statement #1
  input2[j] = output1[2*index]; //statement #2
  }
```

As we see in Fig D.2, one iteration of the interleaver's loop body takes 6 cycles in Design-Trotter, where statement #1 in the code above takes 2 cycles to be performed: it takes 1 cycles to read from memory the constant 2 and `alpha[j]` values, then 1 cycle to write `alpha[j]` to index variable. Statement #2 takes 4 cycles to be performed: 1 cycle to read index value, 1 cycle to perform `2*index` multiplication, 1 cycles to read `output1[j]` value, and 1 cycles to store `output1[2*index]` in `input2[j]`.

In contrast, in Handel-C, one iteration of this loop body takes only 2 cycles: 1 cycle both for statements #1 and #2.

The difference in cycle-budget between the Design-Trotter result and the corresponding Handel-C result will, in most cases, have a overweight of cycles in the Design-Trotter solution. The reason is that assignments and control statements in Handel-C are incorporated in the cycle circuit, whereas they have their individual cycle in Design-Trotter. This means that in practice the implemented solution typically is more efficient in terms of timing performance than the corresponding solution, shown in the Design-Trotter trade-off curves between resource usage and cycle-budget.

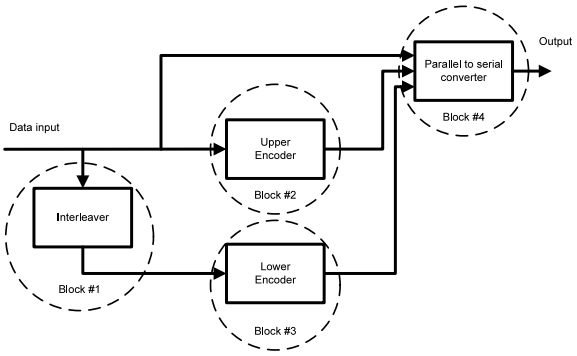


Fig. D.3: Different blocks of the turbo encoder analysed.

If we use Design-Trotter as a guidance tool and want to implement the Design Trotter solution as precisely as possible in hardware, in terms of timing performance, we need to cope with this difference by trying to minimize it as much as possible.

2.3 Timing matching between Design-Trotter and Handel-C

If we consider the nature of these differences, it is clear that the reduction in the cycle budget, in the Handel-C case comes from the assignments. It is not a constant reduction for all assignments but the reduction is more or less equal for the individual assignments whether they are executed in parallel or sequentially.

We can not remove this but try to minimise it. To do so, it is important that all complex code assignments in the Handel-C code should be split into simpler assignments if possible, and that the same code should be used for the Design-Trotter analysis.

3 Examples

In the following we illustrate the methodology applied on the turbo coder part of the HSDPA scheme. The turbo encoder and turbo decoder consist of different blocks which are illustrated in Fig D.3 and Fig D.4 respectively.

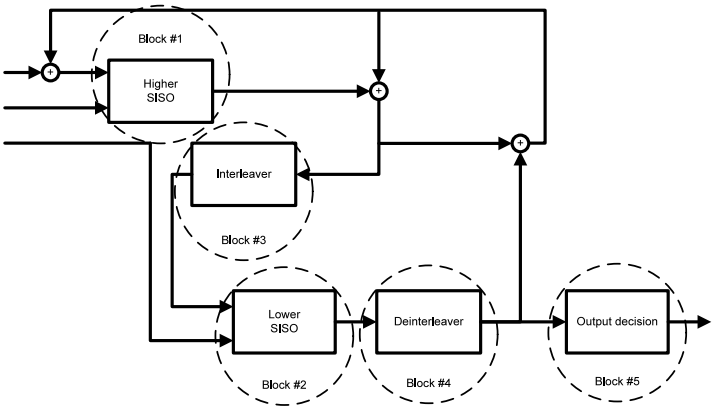


Fig. D.4: Different blocks of the turbo decoder analysed.

Table D.1: Metric of the encoder derived from Design Trotter.

Block #	Block description	COM	MOM	γ
1	Interleaver	0.071	0.786	1.273
2	Upper RSC encoder	0.018	0.667	2.237
3	Lower RSC encoder	0.018	0.667	2.237
4	P/S converter (1 of 2)	0.03	0.636	3.286
5	P/S converter (2 of 2)	0.017	0.525	5.028

3.1 Design-Trotter Characteristics

The characterization results derived by Design-Trotter for the turbo-encoder and turbo-decoder are given in Table D.1 and Table D.2, respectively. The block numbers refer to the numbers shown in Fig D.3 and Fig D.4.

As seen in Table D.1 and Table D.2, all blocks have relatively low COM metric values denoting easily conditioned data-flows, i.e., with almost no nondeterministic control operations in the algorithms. This is due to the fact that the loop indices are almost not data-dependent. The MOM metric values, greater than 2/3, indicate an important data accesses frequency: these blocks require high memory bandwidth in hardware. Finally, we observe high parallelism (high γ value) in the encoder's P/S converter, and in the decoder's SISO processors. It means that these

Table D.2: Metric of the decoder derived from Design Trotter.

Block #	Block description	COM	MOM	γ
1	Upper SISO processor	0.059	0.73	8.276
2	Lower SISO processor	0.059	0.73	8.276
3	Interleaver	0.077	0.846	1.183
4	Deinterleaver	0.143	0.793	1.473
5	Output formation	0.001	0.576	1.405

blocks can benefit from an architecture offering high parallelism capabilities (e.g., FPGA).

Design-Trotter also generates the resource vs. cycle-budget trade-off curves of the turbo-encoder and of the turbo-decoder as shown in Fig D.5 and Fig D.6 respectively.

For the turbo-encoder case, Fig D.5 shows that the most-right solution (3595 cycles) is purely sequential, i.e., at this point all of the turbo encoder operations are performed in a sequential manner using the minimum number of different operation resources.

With the most-left solution (1121 cycles), the maximum available parallelism for the encoder is achieved, where therefore this encoder operates in the fastest way as compared with other solutions. However, this solution is the most expensive in terms of resources. Finally the solutions in between offer different level of trade-off between the most sequential and most parallel ones.

For the decoder case, Fig D.6 shows that there are more solutions than for the turbo-coder. This is mainly due to the fact that there is more processing operations and less data-dependencies, thus more flexibility for scheduling the individual operations.

3.2 The Handel-C implementaion

Different implementations of the HSDPA's encoder and decoder in terms of parallelism and split of complex statements are shown in Table D.3.

The implementation results in terms of timing performance are presented in Table D.4, and the implementation results in terms of resource usage are shown in Table D.5.

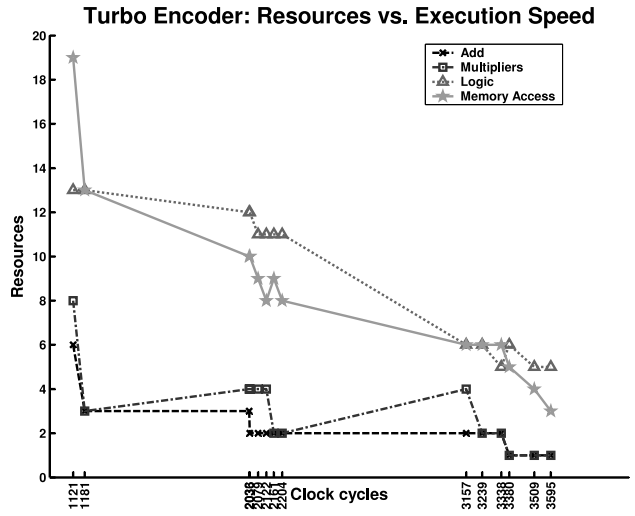


Fig. D.5: Resource vs. Exec. Time graph for turbo encoder.

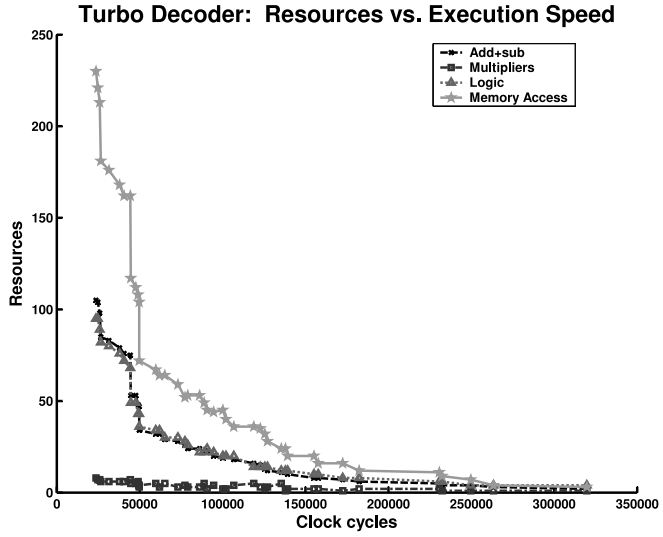


Fig. D.6: Resource vs. Exec. Time graph for turbo decoder.

By examining the results of the different implementation of the turbo encoder, i.e., implementations #1, #2 and #3, from Table D.3, we see the following:

With implementation #1, the original sequential C source code of the encoder without complex statement splitting was used. Here we can observe (Table D.4) that the cycle-budget of this encoder in Handel-C differs from the cycle-budget of the same encoder in Design-Trotter: in Handel-C it is about five times less (3595 cycles / 735 cycles) than the in Design-Trotter.

With implementation #2, all complex statements of the encoder's code, used in implementation #1, are splitted into simpler statements. Here we see that the cycle-budget of the splitted code in Handel-C is now about three times less (5470 cycles / 1880 cycles) than the one of the same splitted code in Design-Trotter. With implementation #3, the internal parallelism is expressed inside all blocks of the encoder's code, used in implementation #2.

At this point we notice that the cycle-budget both in Handel-C and Design-Trotter is reduced about the same number of times (as compared with implementation #2), so using another solution from the trade-off curve gives the same trend in both Design-Trotter and Handel-C implementations.

Let's consider the implementation results of the turbo decoder, i.e., implementations #4 and #5, shown in Table D.4. With implementation #4, the original C source code of the turbo decoder is implemented. With implementation #5, this code is splitted, i.e., all complex statements of this code are broken up into simpler statements.

When comparing implementations #4 and #5, we notice that splitting of complex statements increases the hardware clock speed.

4 Conclusion

In this paper some of the implementation challenges raised by recent and future wireless communication systems have been addressed. More specifically we have described a design methodology for design space exploration and implementation guidance for wireless systems. By applying the proposed methodology to the HSDPA concept we have illustrated its practical usage and benefits. The implementation examples and results

Table D.3: Different implementation used in the example.

#	Algorithm	Solution	Statement splitting
1	Turbo encoder	Purely sequential	No
2	Turbo encoder	Purely sequential	Yes
3	Turbo encoder	Internal parallelism is max exploited	Yes
4	Turbo decoder	Purely sequential	No
5	Turbo decoder	Purely sequential	Yes

Table D.4: Estimated cycle times from Design-Trotter and Handel-C, and execution times from Handel-C implementation, for the corresponding solutions in table D.3.

#	DT [Cycles]	Handel-C [Cycles]	HW clock [MHz]	Exec. time [μs]
1	3595	735	70.1	10.49
2	5470	1880	83.6	22.49
3	4357	1714	83.6	20.51
4	319691	39928	37.5	1066
5	249903	48213	57.6	836.7

Table D.5: Resources use of the different implementations.

#	# of 4-input LUT	# slices	# RAM blocks
1	593	330	6
2	685	416	6
3	597	363	6
4	6348	3547	4
5	5662	3222	4

have shown how the proposed methodology based on our tool DesignTrotter can alleviate system designers from time-consuming and error-prone design tasks, and thus reducing the time-to-market factor. In particular we have discussed the design space exploration and characterization of the turbo-encoder and turbo-decoder for HSDPA and illustrated how the exploration results can be used to guide the back-end implementation phase.

References

[1] E. M. C. Centre, "Overview of 3gpp release 5 - summary of all release 5 features - version 0.10," 2003.

[2] Y. L. Moullec, J.-P. Diguet, T. Gourdeaux, and J.-L. Philippe, "Design-trotter: System-level dynamic estimation task - a first step towards platform architecture selection," *In Journal of Embedded Computing (IOSPRESS)*, vol. 1, no. 4, pp. 565–586, 2005.

[3] Y. L. Moullec, N. B. Amor, J.-P. Diguet, M. Abid, and J.-L. Philippe, "Multi-granularity metrics for the era of strongly personalized SOCs," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2003.

[4] "Dk-design suite datasheet," http://www.celoxica.com, 2007.